Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Scaling of cloud-native IMS network funcions

Scientific Students' Association Report

Author:

Bence Orosz
Viktor Cseh

Advisor:

Prof. Tien Do Van
Csaba Rotter

2022

Contents

List of Figures

List of Tables

Acronyms

3GPP 3rd Generation Partnership Project. 1

AoR Address-of-Record. 11, 12, 45

API Application Programming Interface. 6, 9, 14, 18, 26

CN IMS Cloud Native IMS. 3, 6, 7, 41, 45

CNCF Cloud Native Computing Foundation. 2

CNF Container Network Function. i, ii, 1–3

CPU Central Processing Unit. 5, 6, 8, 9, 13, 18, 26–37, 40, 41, 46

CRD Custom Resource Definition. 18

CRLF Carrige Return Line Feed. 44

CSV Comma-separated value. 17, 18, 21, 22, 29, 38, 41

DB DataBase. 5, 9, 12–14, 18, 45

DMQ Distributed Message Queue. 11–13

DNS Domain Name System. 15

eBPF Extended Berkeley Packet Filter. 19

HPA Horizontal Pod Autoscaler. 2, 3, 5–7, 9, 11, 26, 31–37, 41, 50

HSS Home Subscriber Server. 3, 5

HTTP Hyper Text Transfer Protocol. 12, 18, 43

ICMP Internet Control Message Protocol. 45

ICT Information and Communications Technologies. ii, 3

ID Identity string. 17

IMS IP Multimedia Subsystem. i, ii, 1–5, 41, 43

IP Internet Protocol. i, ii, 1, 2, 8, 9, 15, 20, 25, 45

IT Information Technology. 1

JSON JavaScript Object Notation. 15, 18, 20

K8S Kubernetes. ii, 2

LTE Long Term Evolution. 1

mCPU miliCPU. 28

NF Network Function. 3

NR New Radio. 3

OS Operation System. 1

P-CSCF Proxy - Call Session Control Function. 3, 5, 8

PNF Physical Network Function. 1

PromQL Prometheus Query Language. 20

QoS Quality of Service. i, 1, 2, 26, 27, 29, 31, 39, 41

REST API REpresentational State Transfer API. 9, 14, 18

RFC Request For Comment. 15, 22, 24, 43

RTP Real-time Transport Protocol. 15

S-CSCF Serving - Call Session Control Function. 3, 5

SBA Service Based Architecture. 1

SIP Session Initiation protocol. 2, 3, 5–9, 12–15, 18, 22, 24, 26, 30, 32–36, 40, 41, 43–46, 51

SIPS SIP Secure. 44

SMS Short Message Service. 1

Telco Telecommunication. 38

TSDB Time Series DataBase. 18

TTL Time-to-live. 45

UA User Agent. 44

UAC User Agent Client. 44, 45

UAS User Agent Server. 44, 45

UDP User Datagram Protocol. 12

URI Uniform Resource Identifier. 44, 45

VM Virtual Machine. 1

VNF Virtual Network Function. 1

VoLTE Voice over LTE. 1, 3

VoNR Voice over NR. 3

Kivonat

A konténerizáció jelenleg egy felkapott technológia az ICT világában. A távközlési iparban hálózati funkciókat szoftveresen készítenek és becsomagolnak a konténerekbe, melyeket a felhőszámítástechnikai környezetben Kubernetes kezel. A Kubernetes egy ipari standard a konténereket tartalmazó Pod-ok orkesztrálására (létrehozás, felügyelet, megszüntetés), amely rugalmas és költség-hatékony erőforrás-gazdálkodási (skálázási) lehetőséget ad a változó előfizetői forgalom kezelésére. A Kubernetes beépített dinamikus skálázó megoldása azonban nem veszi figyelmebe a szolgáltatás minőséget, másnéven Quality of Service (QoS). Kutatásra és tapasztalatszerzésre van szüksége az operátoroknak a költség-hatékony beállítások megkereséséhez és QoS garanciák biztosításához.

Jelenleg nincs szabadon használható tesztkörnyezet Container Network Function (CNF) rendszerek skálázására, így mi terveztünk és implementáltunk egy CNF alapú IP Multimedia Subsystem (IMS) környezetet. A megvalósított környezetben tudjuk emulálni többféle forgalmi helyzetet (beleértve a városban lezajló hívás-létesítése a különböző napi időszak szerint) valamint tudjuk tesztelni és össszehasonlítani a különböző üzemeltetési megközelítést (pl. a hagyományos kapacitás tervezési eljárás, automatikus skálázás).

Abstract

Containerization is a trending technology in Information and Communications Technologies (ICT). The industry creates CNFs from those containers and deploys them into Kubernetes (K8S). Kubernetes is the de facto standard for container orchestrations. Kubernetes manages Pods which are the smallest deployable unit in Kubernetes. Kubernetes creates, monitors, manages, and destroys those Pods. With Kubernetes, the possibility is opened to create industry-ready systems that are dynamically scalable on the demand of incoming traffic.

Configuration and validation of the scaling are first needed to achieve the industry's requirements. However, the currently built-in dynamic scaling solution of Kubernetes could not consider the QoS requirements of ICT systems. Research and experiments are needed to help the Operators use Kubernetes and CNFs for ICT systems. To improve the usability of Kubernetes in ICT systems, freely available proofs-of-concept, demos, and test environments are needed.

However, there is no publicly available testbed for scaling CNFs when writing this paper. We plan and implement an IP Multimedia Subsystem (IMS) environment similar to the Operators' environment where scaling experiments can be performed. Then we test CNFs with the traditional capacity planning approach. After that, we test the scaling approach of the built-in dynamic scaling. In the end, we will run an experiment driven by traffic created from a publicly available dataset of customer behavior.

Chapter 1

Introduction

In the last decade, user behavior changed dramatically [1]. Phones were made with better hardware and more intelligent software that enabled customer features beyond Short Message Service (SMS) or phone calls. Video, e-mail, online or offline gaming, and much more features attracted many customers. Those media features have real-time network traffic with strict Quality of Service (QoS) needs [2]. The new media features created the need for the logical separation of the core network and the media services. Those media services were grouped, and an ecosystem called IP Multimedia Subsystem (IMS) was created to provide signaling for multimedia over IP [3]. Soon the Voice over LTE (VoLTE) feature was added to the 4G network to allow simultaneous call, and mobile data traffic [4] [5].

Some parts were dedicated physical hardware called Physical Network Function (PNF). A new trending technology, virtualization, began to rise, which gave the idea that some network functions could be implemented and deployed in a virtual environment instead of the physical network equipment. First, Virtual Machines (VMs)s are used, and the group of one or more VMs is called Virtual Network Function (VNF) [6]. However, there was a problem with VNFs. They are running on a shared server environment called Hypervisor. One server is not strong enough to compete with dedicated hardware, so the solution was to group the Hypervisors into larger compute groups called Clusters to handle the traffic that PNFs served previously.

With 5G, a new era has started with the cloudification process, which was inspired by other fields of Information Technology (IT). The cloud-native systems offer scalability that can expand services on demand, flexibility to operate systems easier, resiliency to make fault-tolerant systems, shorter development to market time, and much more. A new approach was necessary to use the advantages of cloud computing: containerization and building the system from those loosely coupled containers. A container is an isolated group of processes that runs on top of the Kernel of the host Operation System (OS). Compared to VMs, containers use fewer resources because containers use the same Kernel as the host OS. Containers boot much faster than VMs, which means if we could follow the rate of the traffic change with the required running number of containers, then we would save compute resources, which means we could save money as well. That was one of the main reasons that inspired the standardization of Service Based Architecture (SBA) of the 5G core by 3rd Generation Partnership Project (3GPP) [7]. The SBA contains interconnected Container Network Functions (CNFs) that can be scaled up to handle the incoming traffic.

With thousands of containers, the need arrives instantly to manage, control, synchronize and monitor those containers distributed between hundreds of servers. For that purpose,

K8S is the de facto standard currently. It is a container orchestration platform developed by Google, but now, it is a project of Cloud Native Computing Foundation (CNCF) [8].

With IMS, we also see the same tendency of cloudification, where the increasing number of users and demand forces the industry to the scalable functions, where the CNF comes in, which offers the possibility of Horizontal scaling, to dynamically match the user demand and QoS. This type of scaling is only available in Kubernetes, but it is not as ready for an IMS system as it seems initially. This is why we want to try the current scaling operator and find the problems for the future development of a more suitable scaling algorithm that can work on a full IMS and 5G core. HPA is the de-facto solution in the web-development world, but the efficient operation of HPA in the telecommunication environment is not explored yet.

In this TDK, we want to experiment with scaling the IMS systems in this cloud-native environment because we see a rising demand from mobile operators to scale their IMS systems based on user demand and QoS. After all, this Cloud-Native IMS in Kubernetes could run on consumer-type hardware that costs less, and it does not have to run at full speed every time of the day because it can Scale up and down based on the Load and QoS. Nevertheless, we need our small testing environment to test with, and we find that the SIP proxies would be a good start because it has been used in IMS signaling. It would be an excellent start to scale and experiment with that in Kubernetes.

The rest of this TDK is organized as follows. In the second chapter, we discuss the components of IMS and the problems we want to solve. In the third chapter, we present the plan and the implementation of our test environment with the SIP system, the traffic generator and the monitoring component. In the fourth chapter, we carry out benchmarks and scaling experiments with traffic generated according to a data pattern captured by a real operator.

Chapter 2

Problem Description

In this Chapter, we shortly overview a practical problem that motivates our work. We explain the context of operating Cloud-native IMS systems and the need for setting HPA scaling algorithm in operators' environments.

2.1 Cloud-Native Operator's system

These vast and complex ICT systems built from Kubernetes applications are run by operators and used by millions of end users daily. These systems are made from connected CNFs with many moving parts. For a better understanding of building these systems based on Kubernetes, we would like to present a practical example. Once upon a time, Telecommunication Engineers were asked to plan parts of the 5G core or implementation of IMS inside Kubernetes [9], [10].

The IMS provides important services such as VoLTE in a 4G network or Voice over NR (VoNR) in a 5G network. The IMS uses SIP as the signaling protocol for calls. The first Network Function (NF) element of IMS is Proxy - Call Session Control Function (P-CSCF). It acts as ingress and egress for the IMS and forwards traffic to the Serving - Call Session Control Function (S-CSCF). S-CSCF relays SIP messages e.g. Invites to other parties if needed. It also acts as a SIP registrar, which means it handles SIP Registrations with the help of the Home Subscriber Server (HSS) in a 4G network. The HSS acts as a user database, so it stores the required user data for S-CSCF. The architecture of IMS is visualized on figure 2.1.

In creating the CN IMS, the next step is understanding the role of the following Kubernetes components. They are listed in the order in which the inward traffic meets them. The traffic that comes from the users into Kubernetes is first met with the Ingress. The Ingress act as an external load balancer. It distributes the traffic between the nodes. Inside the nodes, Services handle the traffic and load balancing the packets to the Pods. The service knows which Pods are healthy and forwards the traffic only to them. Pods are controlled by a deployment or a stateful set, which belongs to only them. HPA instruments the deployment to scale up or down according to the scaling algorithm. Kubernetes components can be seen on figure 2.2.

With knowledge of the parts of the IMS and the required Kubernetes components, we can start planning the cloud-native IMS. Firstly we need to containerize the software that acts as S-CSCF, P-CSCF, and HSS. After that, we need to create a Deployment, which will create the Pods for the P-CSCF (because it is a stateless application), one Stateful

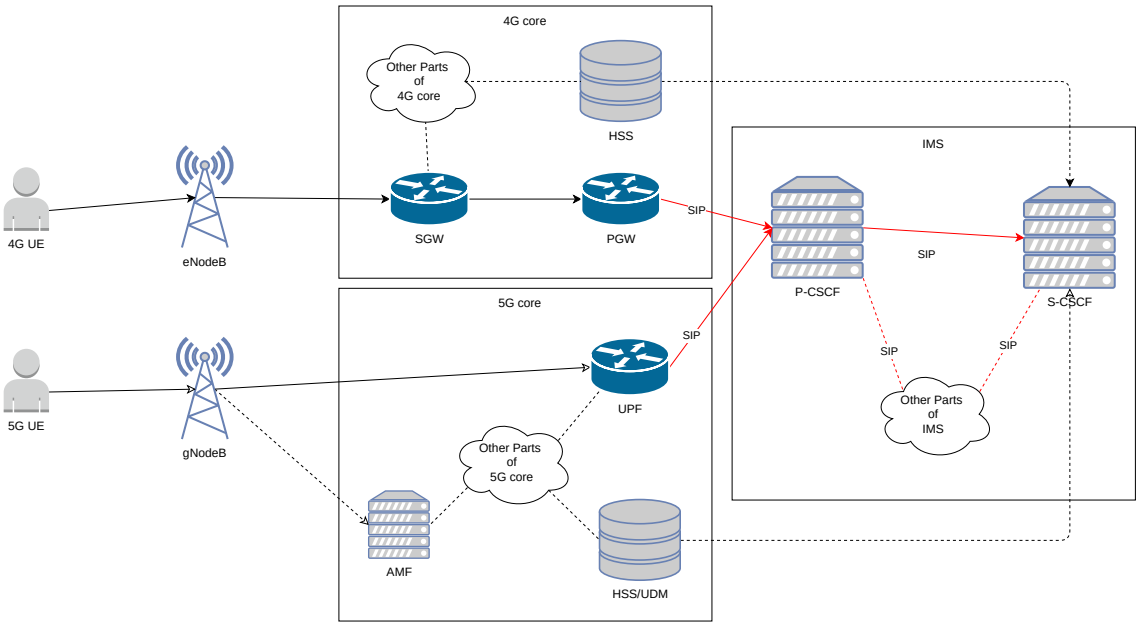


Figure 2.1: IMS in 4G and 5G network

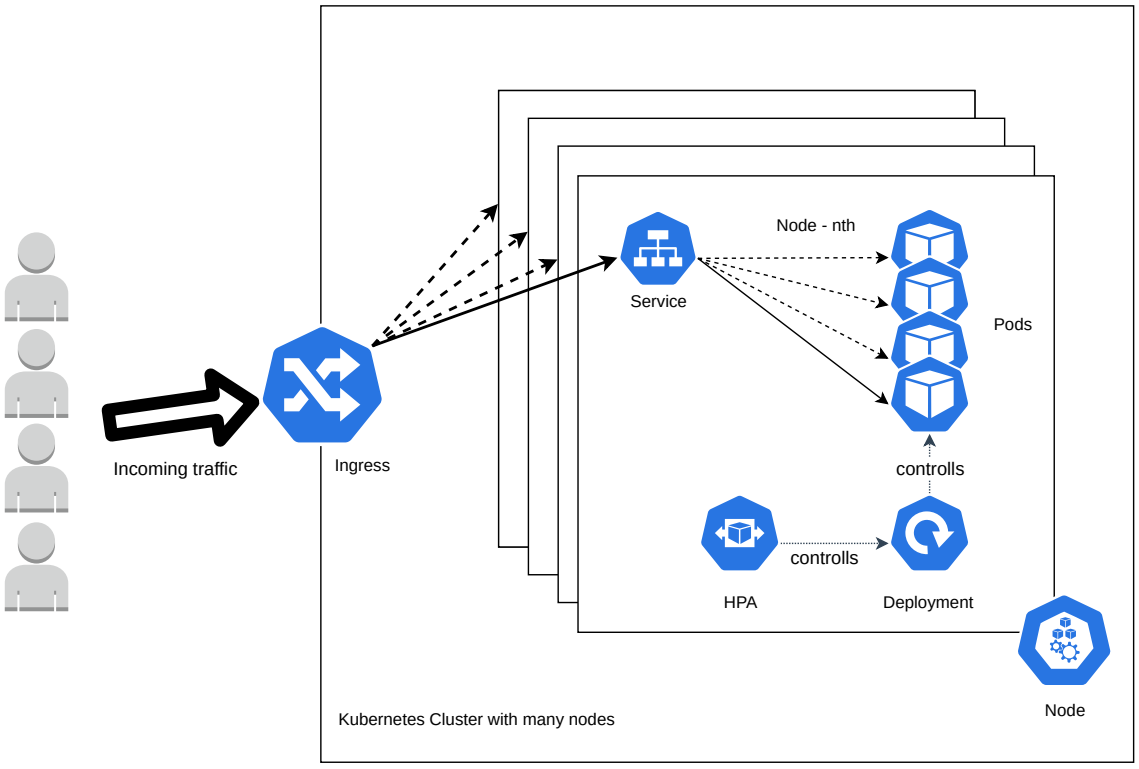


Figure 2.2: Kubernetes components of an Operators' system

Set for S-CSCF and another one for HSS. We might need persistent storage for HSS, so we will mount volumes for the container in the HSS Pod. The next step is to create individual Services for P-CSCF, S-CSCF, and HSS. With that, our Pods can communicate with each other. To make the Service of the P-CSCF externally available, we also need to create an Ingress. The system presented previously is shown on figure 2.3. The arrows are drawn in the direction of data flow from source to destination. Black dashed arrows mean alternative SIP data paths with load balancing. Gray dotted arrows mean Kubernetes control processes, e.g., scaling commands for deployment. Blue dotted arrows mean non-SIP data paths, e.g., User data from HSS. The figure shows the HSS as a single DataBase (DB). However, it is a stateful set with many Pods, but in our IMS-focused perspective, we access it via a Kubernetes Service, which hides the Pods from us. The arrows between P-CSCF and S-CSCF symbolize session-aware load balancing, which means no matter which P-CSCF Pod gets the packet that belongs to the same SIP messages(grouped by SIP Call-id) it will be forwarded to the same S-CSCF.

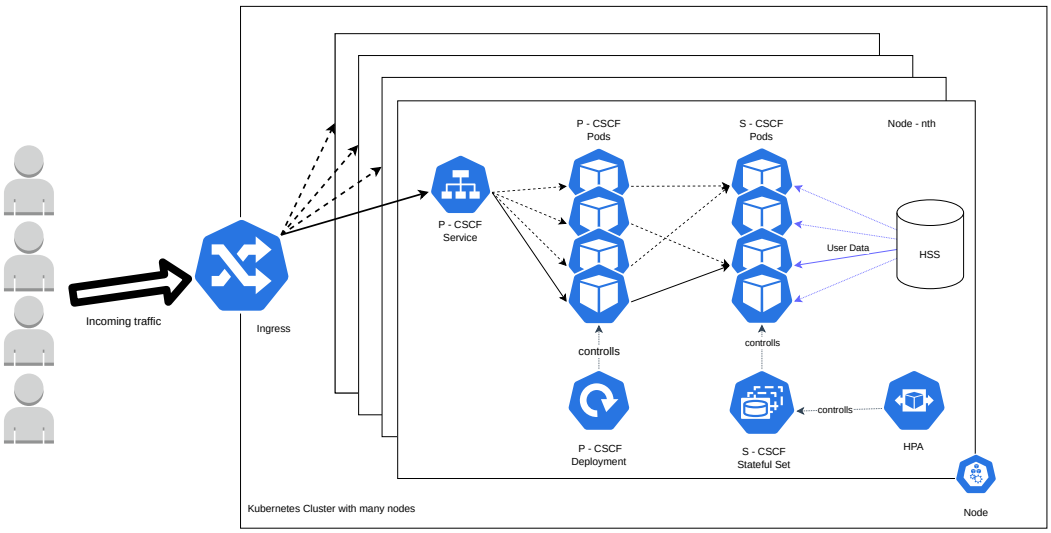


Figure 2.3: Cloud-native IMS in an Operators' system

However, the previous system might operate well we would face an issue sooner or later. The traffic that comes from the users is changing rapidly. We need to plan the capacity of the system according to that. We must choose between two things. Using a fixed number of Pods most of the time would make the capacity more significant than the maximum required. So we would run more Pods than needed, wasting resources and money. If we make the capacity lower than the maximum required, we can sometimes not serve all the users, which violates the contract, and we might lose customers in the long run. We should put some dynamic scaling into the system, which acts as the size of the customers that use the system at that time.

Kubernetes can increase the available resources for the deployed applications horizontally or vertically to help the Operators defeat the challenges successfully. Vertical scaling means adding more power to the Pods, which means increasing the defined Central Processing Unit (CPU) or upper memory bounds of the Pod. Horizontal scaling means increasing the number of Pods for the same purpose. For vertical scaling, the hard upper limit is the hardware on which the Pods run. Because of the hardware limitation and because Kubernetes was also developed with flexible node management (adding or removing physical nodes to the cluster), horizontal scaling with HPA seems a better approach.

HPA is the built-in solution for horizontal scaling in Kubernetes. According to the Kubernetes documentation [11], HPA makes the scaling decision periodically, calculated with the following equation:

$$desiredReplicas = \lceil CurrentReplicas \cdot \frac{CurrentMetricValue}{DesiredMetricValue} \rceil \tag{2.1}$$

Where *CurrentMetricValue* in Equation 2.1 is an average calculated across all healthy pods from the given metric, $DesiredMetricValue$ is a fixed given value from the same given metric set by the Operator during deployment. $CurrentReplicas$ is the number of currently running healthy Pods of a given deployment or stateful set. The right side of Equation 2.1 is surrounded by ceil operator, which means round it up to the nearest whole number. The evaluation of Equation 2.1 at any given time gives us the desired number of pods for the given deployment or stateful set. However, rapid fluctuation might happen in the selected metric value. Kubernetes does not immediately apply for the calculated desired Pod number. Instead, there is a stabilization window in which Kubernetes apply only the highest replica count in that time window. Currently, two types of scaling values are supported. The easier one is the built-in resource metric based, e.g., CPU or Memory utilization. The second one is complicated. It fetches metrics using the built-in `custom.metric.k8s.io` Application Programming Interface (API). To populate the `custom.metric.k8s.io` with data, we should provide an Adapter that queries external data from a data source, e.g., Prometheus [12]. Prometheus is a monitoring system that collects, stores, and makes metric data easily accessible for further use. More on that in Section 3.4.2. The CPU utilization-based metric is far from optimal for scaling in CN IMS because the SIP signaling uses only tiny bursts of CPU, which changes rapidly.

2.2 Motivation and Goals

One of the Operators' open questions is to choose the right algorithm for the horizontal scaling of the Kubernetes Pods. However, a built-in solution exists for scaling, HPA, which was developed for CPU usage-based scaling. According to earlier research [13] and [14] in this field, it is possible to create a better horizontal scaler for the operators' environment than the CPU usage-based HPA.

However, at the time of writing this paper, there is no available ease-to-use, packed working environment to test scaling possibilities for CN IMS. This paper aims to describe that CN IMS environment for scaling, provide a better option for scaling in the operators' environment, and show proof of concept.

Our idea is to measure the response time of the Pod and scale based on that value. The idea seems promising because the length of a signaling message is shorter than the traditional load for which the HPA was built.

Chapter 3

A testbed for scaling experiments

In this chapter, we plan, create and validate the environment needed to experiment with scaling. An overview of the testbed can be seen at Figure 3.1.

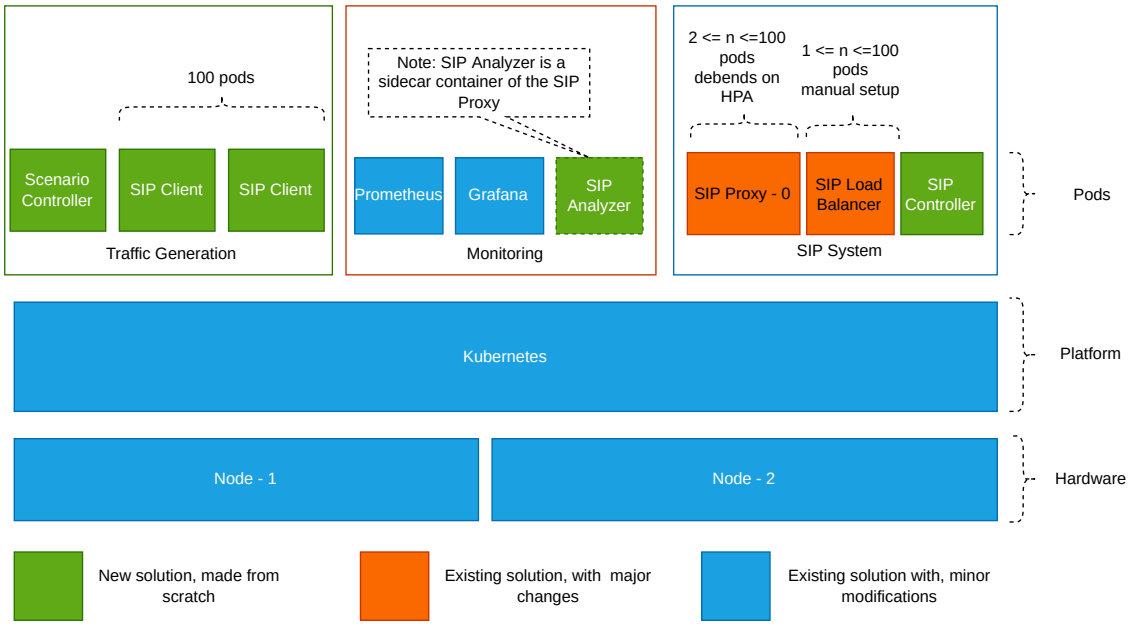


Figure 3.1: Testbed Overview

Planning in Section 3.1 is the first step toward the testbed. The planned testbed has three main parts. The first part is the SIP system which is detailed in Section 3.2. Second is the traffic generator detailed in Section 3.3. The third part is the monitoring that makes scaling possible, more details in Section 3.4. The traffic generator creates traffic based on a real-world traffic shape. The SIP system processes incoming calls while the monitoring collects statistics, e.g., the system's response time. The HPA scales based on the value the monitoring provides, and the feedback loop closes. Last but not least, we close this chapter by validating the testbed in Section 3.5.

3.1 Plan a testbed from CN IMS

The next step is creating the Kubernetes environment where the components can run. We have two physical servers available at the university laboratory to create the testbed.

Also, we wanted to run bare-metal Kubernetes nodes for more precise CPU measurements. With those limitations in mind, we decided to create a single Kubernetes cluster with two bare-metal nodes acting as workers and the first node acting as a Kubernetes master. We decided to generate the traffic inside the same Kubernetes cluster. Further details are in Section 3.3.3. We disabled the ingress for SIP traffic because there is no need to access the cluster externally. We can do that because there is no principal difference between internally originated traffic and traffic arriving through the ingress observed in the perspective of P-CSCF. P-CSCF sees traffic arrive from a routable IP address. The only difference is that it is a private IP address instead of a public one. To eliminate the possibility of fake measurements, clients that generates user traffic are physically separated from the server-side part of the testbed. We have achieved that with the help of Node Selector [15], which restricts the client Pods to running on the first node while the server-side components run on the other node. With that simplification, we removed the complexity of using an Ingress for SIP and the requirement to distribute the incoming SIP traffic among nodes with an external load balancing. Also, we made an environment for the clients to quickly scale them up for more significant load generation like Figure 3.2.

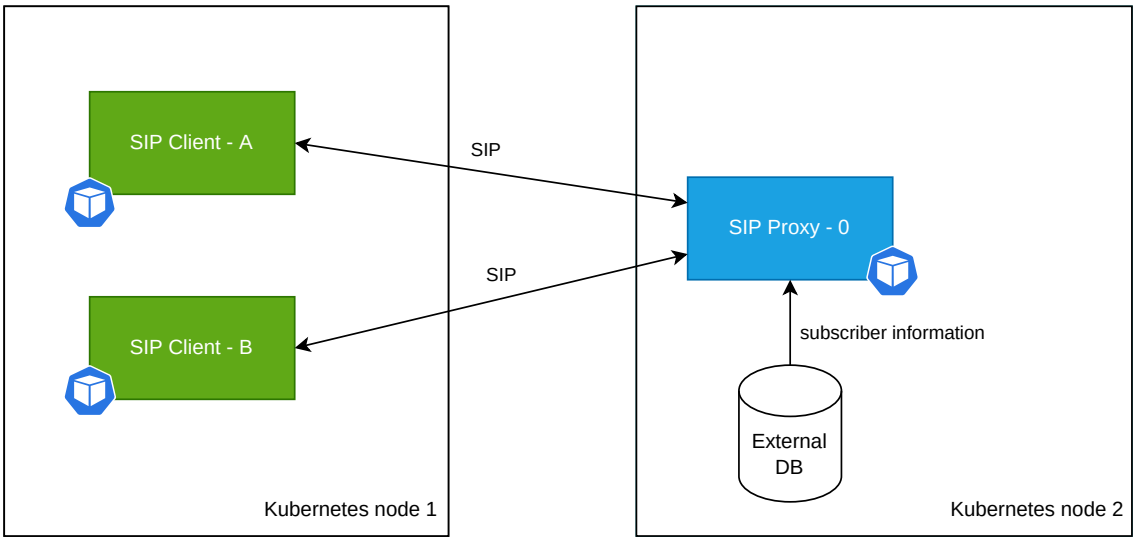


Figure 3.2: Testbed node separation with Node Selector

The next task is selecting the required Kubernetes components for each testbed component. We decided to use a stateful set [16] instead of deployment [17] for the SIP Proxy, as mentioned earlier, it will be a stateful application. More information on that is in Section 3.2.1. Also, we created a service that will act as an internal load balancer for the SIP Proxies. We decided to use a deployment for the SIP Clients for faster boot time. Also, we created a Pod with a scenario-controller deployment to drive the clients as shown at Figure 3.3. More details can be found later about traffic generation in Section 3.3

While planning the Kubernetes components, we faced our first major roadblock. We realized the Kubernetes service could not load balancing according to our needs. Packets with the same Call-ID belong to the same SIP call session, so they must be routed to the same SIP proxy. The default session affinity [18] in a Kubernetes Service is `none`, which means round-robin load-balancing. The other option is source-IP-based session affinity, which is not enough in our case. Clients can change their source IP, for example, when roaming. We realized that there is no built-in Kubernetes solution for that kind of Load Balancing. We made a Pod act as a SIP load balancer called `sip-loadb` to solve the problem. This change is displayed on figure 3.4. The `sip-loadb` is a modified sip-proxy

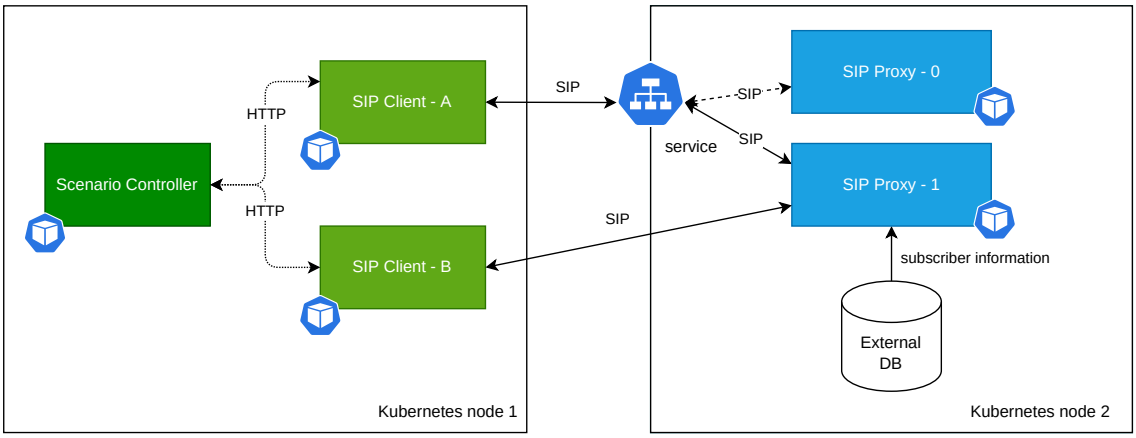


Figure 3.3: Testbed deployments, stateful sets, and services

to act as a stateless load balancer. The Kubernetes Service is moved before the `sip-loadb` with the default round-robin Load Balancing. To work with the new setup, we changed the clients to send the invite message to the service of the `sip-loadb`. The service chooses a round-robin method for the packet's destination, a `sip-loadb`. Then the `sip-loadb` forwards the invite to the proxy selected by the hash of the Call-ID. More details on `sip-loadb` in Section 3.2.2.

However, the load balancer must know the location in the form of IP addresses of the proxies to route the traffic to them. Also, the state of the proxies is needed for the load balancer to know if the proxies are working. To solve that problem, we created another Pod called SIP controller. When a proxy is created or starts terminating, it calls a REpresentational State Transfer API (REST API) endpoint of a Pod called SIP Controller. The SIP Controller notifies the load balancers when the state of the proxies changes. The SIP Controller is responsible for waiting for the graceful periods. To solve the other problem, the SIP Controller manages a DB. The location information in the DB is modified when the state of a proxy changes. The Load Balancer refreshes that information from the DB when the SIP Controller orders it. Gray arrows mean control messages on the Figure 3.5. More information on SIP Controller is in Section 3.2.3.

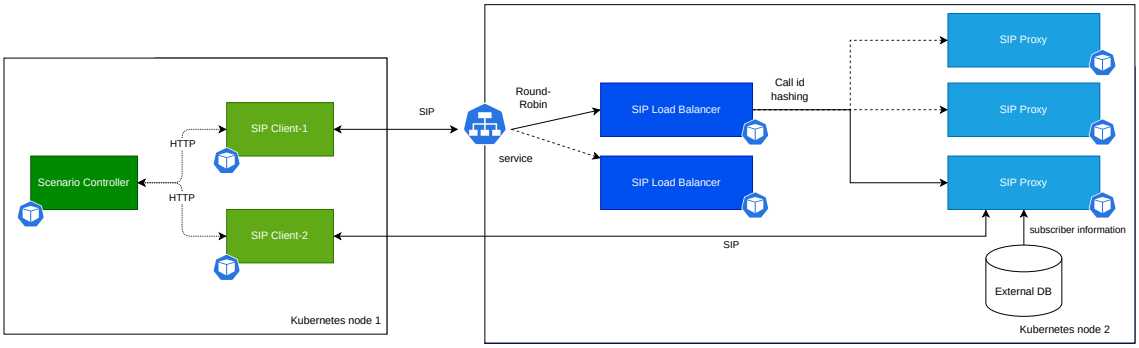


Figure 3.4: Call-ID based load balancing

Finally, we reached the point when we could talk about scaling. For scaling the sip-proxy, the HPA is used. HPA makes the scaling decision based on CPU usage or on custom metrics provided by an Adapter that gathers data from an external source. That piece is added on the Figure 3.6. More details on scaling are in Chapter 4.

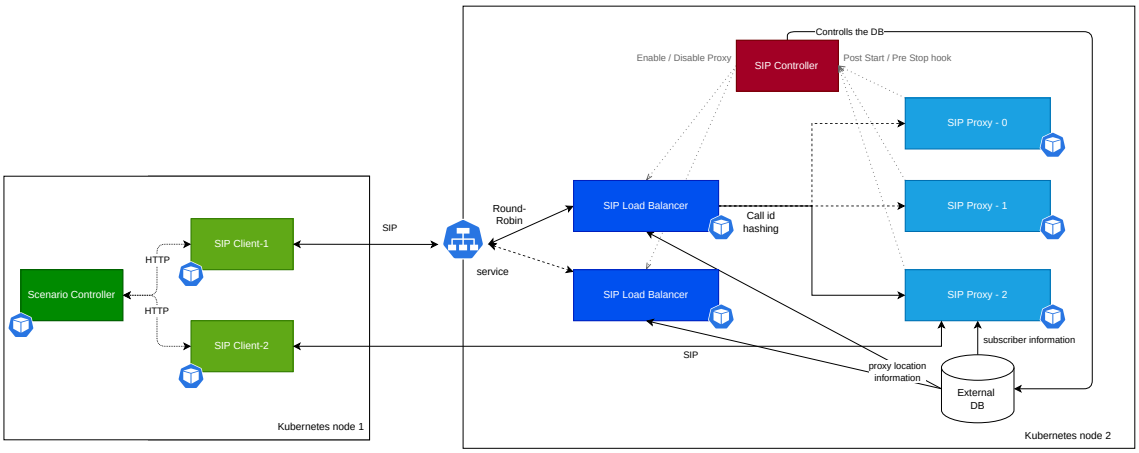


Figure 3.5: Testbed with SIP Controller added

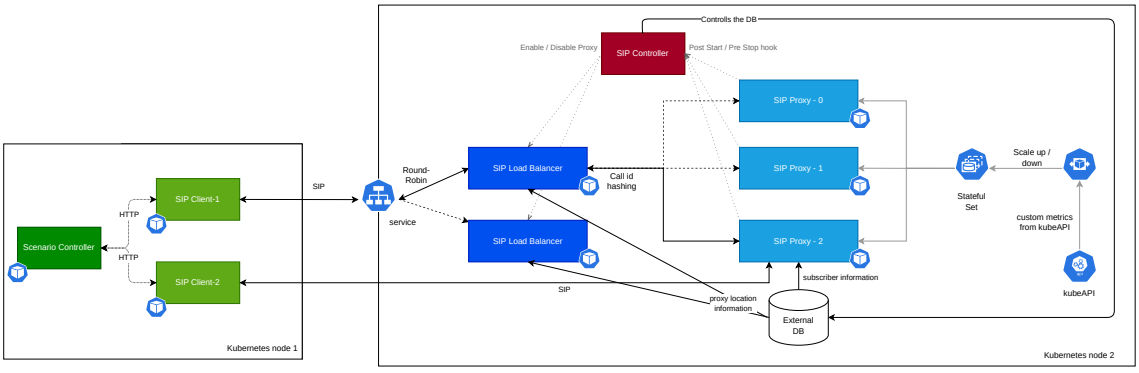


Figure 3.6: Testbed with HPA added

However, we are not done yet because we forgot something. When the HPA scales the Proxy up, its location table is empty. No Address-of-Record (AoR) records exist, so the proxy can not route invites to the clients. To solve that problem, when a new sip-proxy starts, it sends a Distributed Message Queue (DMQ) message to a proxy discovered via the DMQ service to ask for the content of the in-memory database, which holds the information of the registered clients. More information on DMQ in Section 3.2.1. On Figure 3.7, the new DMQ service is added.

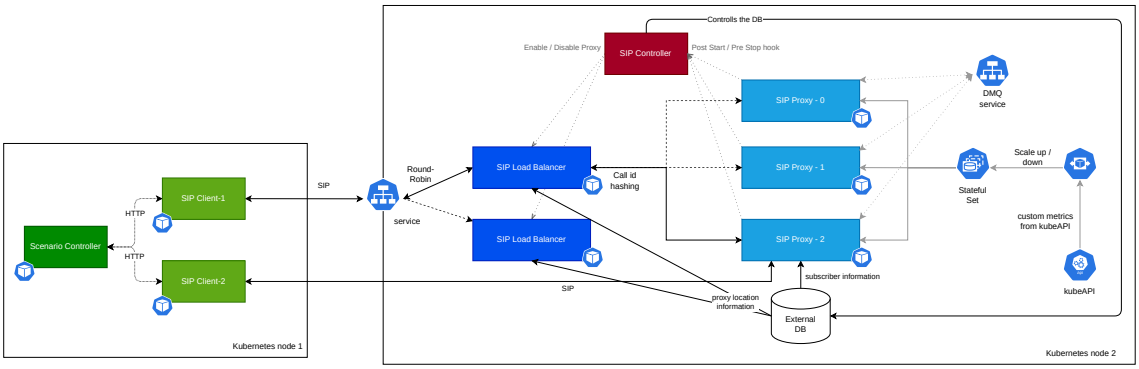


Figure 3.7: Testbed with DMQ service added

Finally, we finished planning our testbed. After implementing each component, we packaged the project into Helm Charts to make the testbed deployment easier. Helm is the de facto package manager for Kubernetes. We created multiple packages, known as Charts, for the deployment process. It is possible to populate the Charts with variables defined in deployment time with the built-in templating feature. The `{{expression}}` syntax is used to evaluate and substitute an expression at deploy-time. More details can be found on each testbed component in the following sections.

3.2 SIP System

In this section, we want to talk about the SIP environment we use for testing and how it works. First, we want to talk about the SIP Proxy in the Section 3.2.1, how it works, and how we check if it is ready for SIP messages. Second, we talk about the LoadBalancer and how it works when the number of Proxies increases and decreases. Then we talk about the SIP Controller, how it controls the load balancer based on how many Proxies there are, and watches if the Proxy is ready or not.

In the following subsection, we want to talk about load Generation, how we designed the clients to be independent of the scenario and what controls them based on the scenario file. First, about the Clients in the Section 3.3.1, next about their controller in the Section 3.3.2, and third, about the scenario file generation, which the scenario controller works from.

3.2.1 SIP Proxy

The SIP Proxy is one of the core components of the testbed. As mentioned earlier, it is responsible for handling Registers and Invites. We decided to use Kamailio as the SIP software. It is an open-source, well-documented, mature project with many pluggable modules, including stateless and stateful message handling, authentication and authorization, support for external DB connection, and many others. A Docker image is also available at DockerHub [19]. On the internet, many tutorials are also found, e.g., Nick vs Networking [20].

Firstly, we created our own Docker image, available at Section A.3 using the original image as a base. We mainly added debug tools, e.g., `tcpdump`, `iproute2`, `netcat`, etc. We also created a symbolic link to `tcpdump` to fix a temporary bug related to `ksniff` [21] and its built-in `tcpdump` upload. We also created a bash script that runs on the start of the container and replaces placeholder text with the value of environment variables, and starts Kamailio. At the end of the script, there is a clever trick. The command `tail -f /dev/null` will run forever, which means that after the crash of the Kamailio application, it will not terminate the container, which would cause a restart. In production, we would love it if our application would restart itself in case of a fatal error. However, in a laboratory environment where we test the software limits, we are interested more in the exact error message that a fatal error would create, or we would like to enter into the container and investigate the situation after a crash. The entry point script is available at Section A.4.

Secondly, we created the config of Kamailio available at Section A.5 of the SIP Proxy. We started with the example config and made modifications to make it capable of acting as a Registrar and handling Invites statefully. We decided to use DMQ to synchronize AoRs among proxies. We changed the User Datagram Protocol (UDP) port for DMQ messages to 5061 from the default 5060 UDP port. With that, we separated the DMQ traffic from the measured SIP traffic. When a Register successfully creates an AoR, it will send that information to the other proxies. When a new SIP Proxy starts, it will request AoRs from another already running Proxy. With that, we opened the possibility to scale the SIP Proxies.

We created the stateful set available at Section A.6. We added the required environment variables as well. Also, we added a readiness Probe which means the Kubernetes will send Hyper Text Transfer Protocol (HTTP) requests to SIP Proxy. We added a module called `xHTTP` that can handle those HTTP messages. Creating a readiness probe is required to

know when the Proxy is finished booting and able to serve requests. We also created the Kubernetes service for DMQ.

Finally, we created the Helm Chart, and all the configs will be attached as config maps available at Section A.11 to the Pod at deployment time.

3.2.2 SIP Load-Balancer

The load balancer is the next component to discuss after SIP Proxy. Its most important responsibility is to distribute traffic between proxies based on the hash of the Call-ID. This component uses the same Kamailio Container image presented at Section 3.2.1.

The SIP Proxy presented at Section 3.2.1 is stateful because it fulfills the definition from Table A.1. We had two options for choosing the type of load balancer. Use a stateful proxy as a load balancer or a stateless proxy as a load balancer. If we choose the stateful option, we would use CPU time and increase the response time of the overall system to handle all the things that come with transactions. If we choose the stateless option, we would ignore the transactions and forward all the packets with more straightforward logic.

Firstly, we decided to use a stateless solution because we maintain the transactions with forwarding based on the hashing over Call-ID, which is persistent across all transactions in a dialog.

We created the Kamailio config available at Section A.8. The config is much simpler due to the stateless behavior of the load balancer. The two main parts of the config are error handling plus sanity check and the dispatcher logic. The load balancer, due to the behavior of the Kamailio dispatcher module, can load the list of the existing destinations from a file or an external DB. We decided to use an external DB because it is more scalable than a locally available file.

Thirdly, we also created the deployment available at Section A.9 of the load balancer and the service available at Section A.10 with round-robin mode. We provide the Kamailio configs for the Pods as we did at the SIP Proxy.

Finally, we created the Helm Chart similarly to the SIP Proxy. However, we have not fully solved the question of load balancing. We have not said a word about when we should route or not route traffic to a Proxy. In the Dispatcher module of Kamailio, there are two ways to change the state of a destination. The first method is the active probing mode, which periodically probes all the proxies, and after k times a successful response, the load balancer allows traffic to that destination. After x times a failed request, it disables that proxy. The second method is the manual mode, which means we can manually adjust the forwarding rule. The first method might sound better but think about it with the Operators' heads. The Operators, for example, need maintenance time to manually disable proxies on specific nodes. Also, if we think on a bigger scale, the load of active probing will increase significantly and burn the critical and expensive CPU time, reducing the system's capacity. However, we automate the manual mode if we create a controller that turns on or off proxies with a given rule. The automated manual method is what we have chosen. More information is in the next section Section 3.2.3.

3.2.3 SIP Controller

SIP Controller was born with the need to control the load balancers according to the lifecycle of the proxies. There are two built-in Kubernetes hooks to manage the lifecy-

cle of Pods. The Post-Start hook is called immediately after the container is created. The Pre-Stop hook is called after the graceful period of the container and immediately before the container is terminated. The approach means if we create an application, e.g., a web server with REST API, it can handle those hook calls and control the load balancers. We created that API endpoints to contain the hostname of the proxy, e.g., `/proxy/enable/sip-proxy-0` where the hostname is a variable. Go programming language was used to create Kubernetes, so it is not surprising that Go has a powerful Kubernetes library. We use that to create the SIP Controller.

We added another responsibility to the SIPs Controller. After the boot, it prepares and populates the DB for the Proxies and load balancers. We thought it fits logically better into the server-side SIP controller than the client-side scenario controller. It inserts into the DB all the pre-generated users from id 0000 to 9999 and the proxies from 0 to 99. Those numbers can easily be adjusted if needed.

We also added a helpful feature to the controller. A REST API endpoint exists that collects all the registered client ids from the proxy-0. This information is used in the SIP Scenario Controller.

3.3 Traffic Generation

3.3.1 SIP Client

We wanted a small and scalableSIP client that observes theSIP RFC, so we wrote a program that can be controlled from a central point (Scenario Controller) and startsSIP calls on command. Hence, it is independent of the scenario. We also put it in a Kubernetes environment to scale it quickly if we want more traffic to hit the proxies. Up to 100 clients can send and receive invites simultaneously inside one Pod. We need this because, on one Kubernetes node, there only could be 250 Pods which cannot create nearly close enough traffic that we want to test the proxy with.

BecauseSIP is the industry standard for signaling. There are many implementations. We tried a couple of them, but either it was not written to be scalable or a big spaghetti code without documentation, so we decided to write our client in Go.

Why go? It is a lightweight, compiled, c-like, not object-oriented language, so everything is given for writing a lightweight, scalable, fast program so that we can run multiple instances at once without overloading the server as seen on Figure 3.9 and Figure 3.8.

First, the go library we tried to use for theSIP client was faulty. It can not handle the Call-IDs and branches correctly, so after registering to the proxy, the call would not be received by any clients because of the faulty branches [22].

We decided to use the second library, but it was not bugless. We had to fork the git repository and fix the `Via` handling in the library because the writers did not think someone was trying to use their library with multiple hops. We had to fix the authentication header's building because if not all fields had values, it would have built a faulty package and sent it to the proxy [23].

The invite response was the last thing we had to fix because it would not change the source and destination address.

The Client we implemented could be controlled on a WebSocket connection. The controller sends a JavaScript Object Notation (JSON) on the connection, which the client can parse and act according to, for example, register on the proxy and send an invite to the correct Pod. It is all we need because we want to simulate how a call would be established and do not need the actual Real-time Transport Protocol (RTP) connection to be created.

3.3.2 SIP Scenario Controller

The scenario controller controls the clients based on the scenario file generated by the Jupyter notebook.

The implementation to control the clients with a centralized controller was based on, that we, as observers, do not know how many clients (client Pods) there will be and how we can reach them. So we figured out that it would be easier if all the client containers connect to a controller on WebSocket, so we know precisely how many of them there are, and through the WebSocket, we can send them commands on what to do. In contrast, we do not have to know all clients' IP addresses. Only the clients must resolve the Scenario Controller's IP by Domain Name System (DNS) and connect to its WebSocket.

The Scenario Controller makes the clients independent from the scenario, and they do not have to know what they will do in the future.

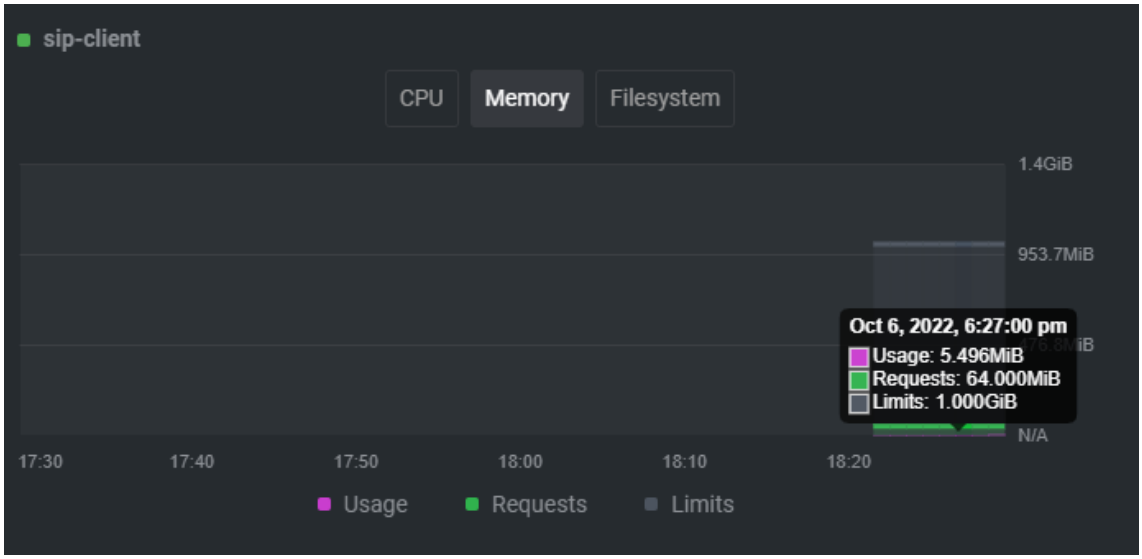


Figure 3.8: SIP client's memory usage

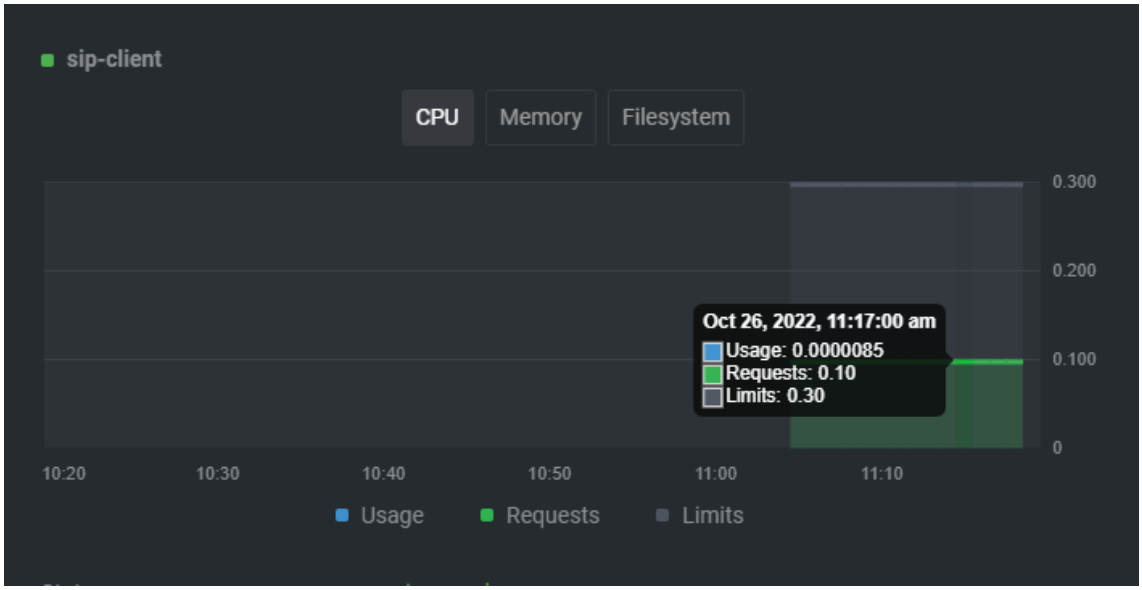


Figure 3.9: SIP client's CPU usage

These CSV files are inside the scenario controller's container, so we have to select which scenario we want to run on its control page, and the controller starts sending the control messages seen on Figure 3.10.

We wrote the controller to be fast and easy to use. That is why we also choose the go language for this, to keep up with the controlling of more than 1000 invites per sec, and we can also implement a small webpage inside it, where the operator can start the scenario on a webpage by selecting which one he wants to run efficiently.

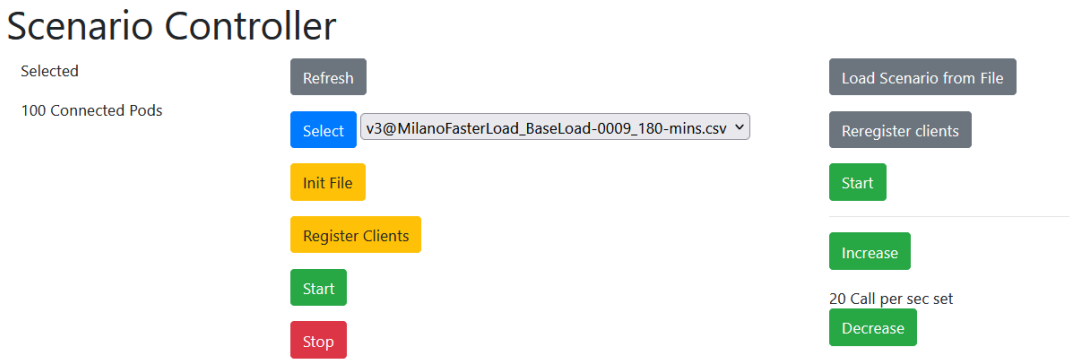


Figure 3.10: Scenario Controller's user panel, where the Register messages and Scenario selection and Start can be controllerd

3.3.3 Input generation for the Scenario Controller

As mentioned previously in Section 3.3.2, a CSV file is used to drive the Scenario Controller. This CSV generator is a Jupyter notebook that creates the CSVs in offline mode. Then the generated CSVs are uploaded into the scenario controller during deployment time.

The output CSV uses the following format:

```
caller;callee;wait_before_next_in_ms;seconds_since_start;arrival_rate_in_s
4600;4700;1000;0;1
7000;7100;1000;1;1
9400;9500;1000;2;1
8600;8700;1000;3;1
7600;7700;1000;4;1
3000;3100;1000;5;1
3200;3300;1000;6;1
9200;9300;1000;7;1
[...]
9000;9100;1000;1797;1
1200;1300;1000;1798;1
1400;1500;1000;1799;1
```

The first line is the header of the file. The following k rows are interpreted line by line, and each line contains the properties of one call. The meaning of the parts, each separated by a semicolon in order, are the following: the Identity string (ID) of originating client of the call, the ID of the terminating client of the call, the time in milliseconds that the scenario controller should wait before sending the following command, the seconds elapsed since the start of the scenario and the arrival rate ($\lambda(t)$) in that second.

3.4 Measurment setup

After creating the testbed at Section 3.1, the next step towards scaling is to create measurements and collect the data for scaling. We have two main goals to solve. Firstly, we need an environment prepared for what we can use to validate the testbed (more on validation in Section 3.5) and collect data for debugging. Secondly, we need online, almost real-time data processing for scaling. Those two different use cases are illustrated on Figure 3.11.

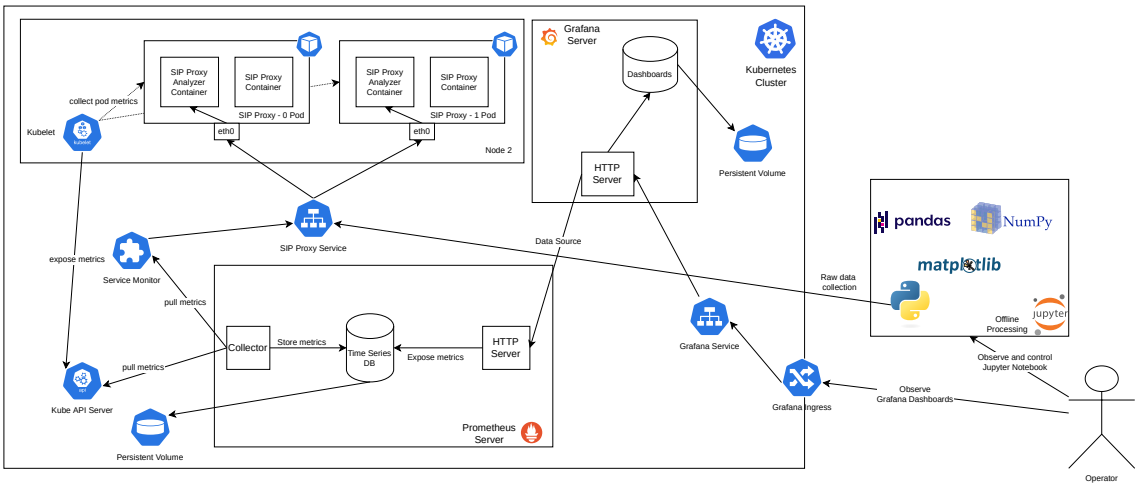


Figure 3.11: Measurement Setup with 2 use-cases

Two important notes for Figure 3.11. First: Although only Node 2 is displayed, all the other components inside the Kubernetes Cluster also run on nodes. The node on which they run is not essential. It can be any available node. For the components inside the Node 2 box, it is mandatory to run on that particular node. Second: Although Prometheus Server and Grafana server are displayed as logical boxes, they consist of pods, configs, services, and many more Kubernetes components. We do not want to focus on them now for simplicity.

The first and most crucial component is the SIP Analyzer (more on that in Section 3.4.1) side-car container in the SIP Proxy pod. This captures traffic and, from that, calculates response times. Then the calculated response time is stored in internal storage, which has two external interfaces. The first is a REST API which creates JSON output. The REST API is for short scenarios and only for offline local debugging purposes. The second and more important one is also a REST API, but it returns or exports data in a format Prometheus can process. The program is why it is also called a Prometheus Exporter. Prometheus contains a Time Series DataBase (TSDB) for storing the collected data known as metrics. Prometheus uses a pull architecture, which collects data periodically from resources. Service Monitor Custom Resource Definition (CRD) is used to discover pods exposed with a Kubernetes service. Prometheus also collects Kubernetes internal metrics, e.g., CPU usage from the Kube API. On each node, the Kubelet, also responsible for pod scheduling, collects metrics from the pods that run on that particular node and exposes those metrics to Kube API. Prometheus exposes those metrics with a HTTP server and makes it possible to run queries with calculations on those time series data. That is where Grafana comes in handy to use Prometheus as a Data Source. Grafana can run Prometheus Queries and visualize those results in real-time on customized dashboards, e.g., Figure 3.12. The data from Grafana can be exported to CSV files for long-term

storage and later further processing. We also use Python's data-science tool pack, e.g., Pandas, NumPy, Matplotlib, and Jupyter, for offline processing.

Figure 3.12: Grafana dashboard during scenario

3.4.1 Analyzer

The analyzer is a go program we designed to run in a sidecar in the proxy's pod and watch the incoming traffic. We use the `gopacket` library to monitor interfaces and give a structure to handle packets efficiently. The interface listening is possible by the `libpcap` that the lib uses and sped up with Extended Berkeley Packet Filter (eBPF). Side-car means it is another container in the Kubernetes pod, and because it is in the same namespace as the prox, it sees all the incoming and outgoing packets, which makes it the perfect solution for this task.

This way, with this program, we can capture the incoming and outgoing packets, which we can stand in pair by the `Call-ID` and calculate the difference in time as their arrival and forwarding.

Let us say an invite arrives at the proxy. We record the packet type, then search if its pair is already recorded. If yes, we calculate the response time. If not, we store the packet's essential aspects in a map to be easily searchable by `Call-ID`.

We are interested in 3 packet pairs in this measurement. The first is the unauthenticated invite and its response time. The Second is the authenticated invite, which takes time to forward to the called client. The third is the `200 OK` that the called client sends to the caller, which the proxy has to forward, as shown in figure Figure 3.13 above.

These recorded response times are stored in two ways. One is the Prometheus exporter, where we record how many packets arrived and left and the response time. In the exporter,

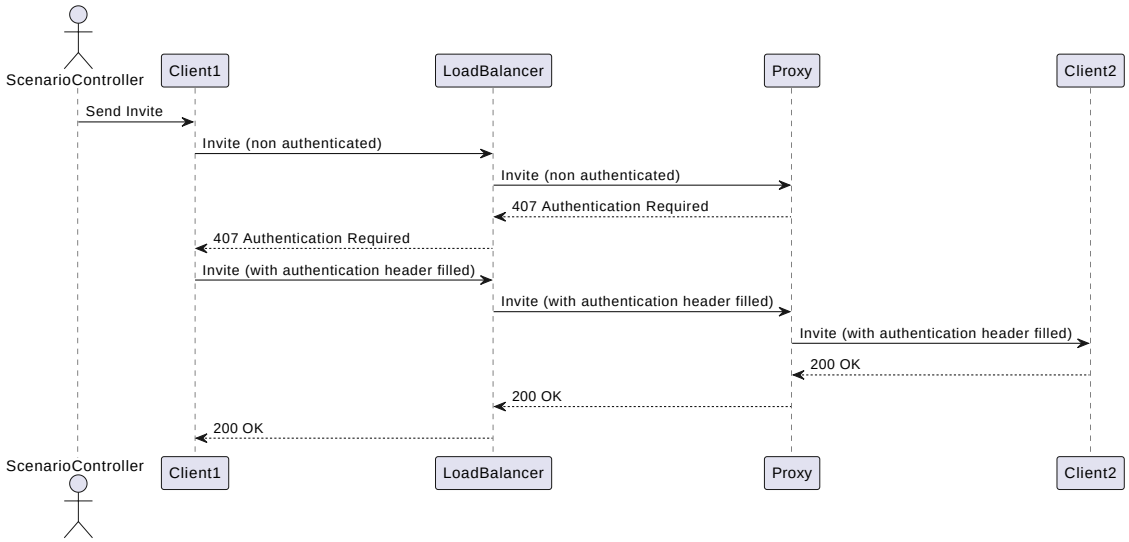


Figure 3.13: Sequence diagram of a call

we only note the response time in buckets, which is explained in the following subsection. The second way is to record in memory, so with a request at an endpoint, it gives all the recorded response times back in a JSON format, which we can use to analyze how good or bad the scaling algorithm was and to create a visualization of the changing of the response time.

3.4.2 Metrics Collector

The Collector we use to get the measurements from the proxies is Prometheus. It is the industry solution to get data about servers/services/apps and pods in Kubernetes. It has a built-in feature to work with Kubernetes, so we do not have to implement a way to get the proxy's IP and measurements in real-time. It is already implemented. The only downside is that it "scrapes" the pods every 30 seconds, so the data flow is delayed by this a bit, which in the real world is not much, but when we use it in sped-up scenarios, it could delay the scaling of the proxies. The other downside is that it uses buckets, which show the limit of the single response time. So example, if the response time was 13ms and we have the buckets of 1, 2, 4, 6, 10, 14, it falls into the buckets that are smaller than the response time, in our case, into 1, 3, 4, 6, 10, so we only know the distribution of the data. The data is enough to compute the 99th percentile. We want to scale based on that.

Working with this is beneficial because it reduces the overhead on the clients, making long scenarios possible to compute in almost real-time.

3.4.3 Visulaization

For visualization, we use Grafana. It is a tool to display data from Prometheus. It runs a query written in Prometheus Query Language (PromQL) to get the desired values in almost real time.

If we see the scenarios as real-world tests, it is like what an operator sees from the incoming traffic to its system.

Prometheus and Grafana are excellent tools to monitor as an operator, but they are not meant to use as long-term storage and visualization tools. It has 15-day retention, so if we want to store the scenarios, we must find another way to store them. For this, we implemented the second approach on the analyzer to collect the results in JSON format. When the scenario ends, we curl the endpoint and convert its return into a CSV file, which we can store long-term and visualize and analyze using a Jupyter notebook for validation and processing.

The Jupiter notebook uses Matplotlib, pandas, and NumPy for data processing and visualization. These results can be seen in Section 4.1

3.5 Validation

In the previous chapters, we talked about our results, but for those measurements to be valid, the test bed has to be validated.

Validation of this distributed system is not a simple task if we want to do it as one big system. However, suppose we slice it into four logical parts and validate the logical parts by themself. In that case, we can easily validate the system because it is a separate system that only gets input from those components.
The first part is the scenario file because the scenario controller sends commands to the clients based on this file. Second the Scenario Controller. It needs to send the correct command to the correct client at the given time, as the scenario file says. The third is the client. It has to follow the SIP RFC, send out only the required packets without error, and handle the incoming ones as the RFC requires. Fourth is the analyzer because even if everything is correct above, we cannot record the proxy's or load balancer's response time. If it is not accurate, then the whole measurement is invalid.

3.5.1 Validation of the Scenario File

A Jupyter notebook generates the scenario file, but it needs to be checked to be sure it implements what we wanted. For example, the first generated files did not follow the convention that the even pod number calls the one higher odd-numbered pod. The pod numbering is a 4-digit number. The first number, 2, is the pod id, and the second 2 is the softphone's number. In the first version, the Jupiter notebook has not added a 0 padding for the pod id, so for example, the 7 ID's pod was 700 instead of 0700.

In the first implementation, there was a simple bug. For example, if the scenario controller got the caller id 711, it parsed it to pod number 71 instead of pod 07.

This bug was discovered when we looked at the traffic recorded by Wireshark at the proxy for debugging purposes because the sip load was not what we expected. As it turned out, the client IDs were not what they should be, so we started investigating where it went wrong and saw this bug.

To validate the CSV file, we looked at the generated CSV to check the pod and client IDs matching when we found that the IDs were incorrect. We also checked that the time between the two invites is generated according to the load. For example, if the load is 1, the file says to wait 1000ms before sending the following invite command.

After finding and fixing this bug, we tested the scenarios by running them and watching the output of the Scenario controller with Wireshark and logging.

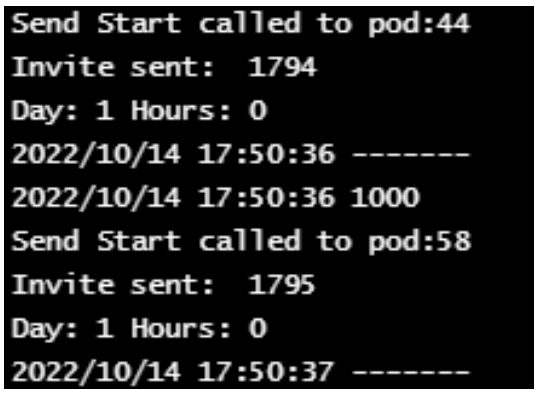
3.5.2 Validation of the Scenario Controller

The scenario controller's job is to control the clients according to the scenario file. We validated the scenario file, so next is the controller.

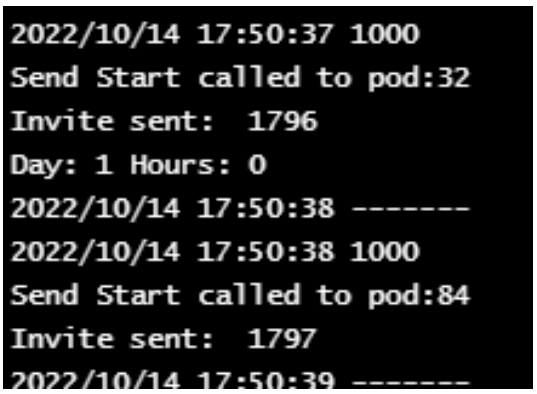
The easiest way to validate the controller was to watch the outgoing commands to clients to validate that it worked the way we designed it. We first added logging to the controller, so we could see on its STDOUT what the controller wanted to send out and compare it with the Wireshark capture and that capture with the scenario file to see if the density of the invites is equal in all three places.

```
ten_minute;milliseconds;calls_in_ten_minutes;caller;callee
0;0;600;7200;7300
0;1000;600;0400;0500
0;2000;600;5600;5700
0;3000;600;6400;6500
0;4000;600;8000;8100
0;5000;600;0000;0100
0;6000;600;3000;3100
0;7000;600;6800;6900
0;8000;600;7600;7700
0;9000;600;4000;4100
0;10000;600;2400;2500
0;11000;600;0800;0900
0;12000;600;9000;9100
```

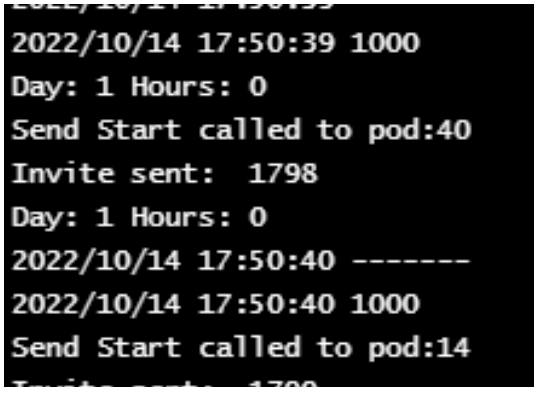
Figure 3.14: Beginning of the Load1.csv



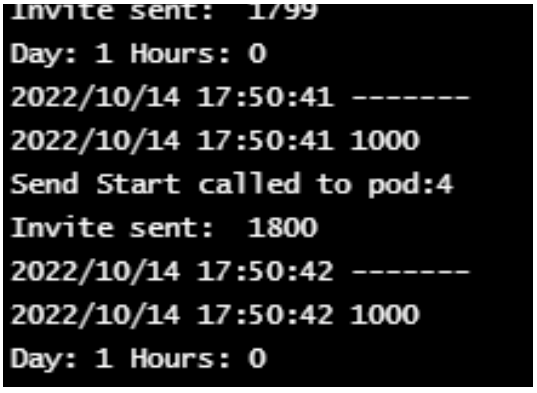
(a) First part of the log



(b) Second part of the log



(c) Third part of the log



(d) Fourth part of the log

Figure 3.15: Scenario Controller log

After the comparison, we saw that the controller sends the correct command at the correct time to the correct client.

The problems were not with the control. Only in the development of the controller we experienced a couple of bugs, which could be fixed in no time after its discovery. Figure 3.15

3.5.3 Validations of Clients

It is essential to send the correct invite packet at the correct time and respond correctly to the invite and the register messages to create the required load. We had to validate the clients.

For validation of the clients, we used Wireshark. We captured the incoming and outgoing sip messages on the clients and load balancer. On proxies, we can see that the invites are sent according to the scenario file, so the difference between the sent invites is precisely what the scenario file declares to be. It follows the sip RFC, so everySIP packet file is filled correctly. We had to validate the library that we wanted to use for the clients because it is a third-party open-source library [24] and to be sure that it also follows theSIP RFC, its validation was essential.

The first library failed its validation, so we had to search for a new one, where we chose the Kalbi project [24]. This library was not perfect either. We had to fork it from GitHub and make some changes, for example, fixing the via handling.

Before every test, to be sure all clients are registered and begin working as they should, we always checked on the proxy with the `kamctl ul show | grep Records` command, which will give back the number of registered clients. After we see the correct number, we start the scenario on the controller.

3.5.4 Validations of the Analyzer

As the most important slice of the system, we had to validate the analyzer. It has to be the last one to validate because it is the most important out of them, and to be 100 percent sure that the validation is correct, we had to leave it last.

Before validating the other elements, we tried to work with the analyzer. However, it never showed the expected measurements because the bugs in other components made the results invalid, and we thought it was the analyzer's fault, but it was not.

We validated the analyzer by recording the incoming and outgoing packets with Wireshark on the proxy. After the test, we compared the .pcapng with the recorded data by the analyzer, and we found it valid.

As the 2 picture shows on Figure 3.16 and Figure 3.17, it logs out the second received packet from the pair and the ms difference between the pairs. As shown in the pictures, the logged difference can be validated by Wireshark.

Even if there are a couple of jumping response times, the Wireshark capture received it the same way, so even if it is strange and looks like some bug, the captured packets show the same.

For testing the calculation, we programmed a debug mode into the analyzer. An environment variable can turn it on. After turning it on, it will write the currently calculated packets to the STDOUT, and we can validate by hand that the correct packets are calculated. This log can be seen below in Figure Figure 3.18

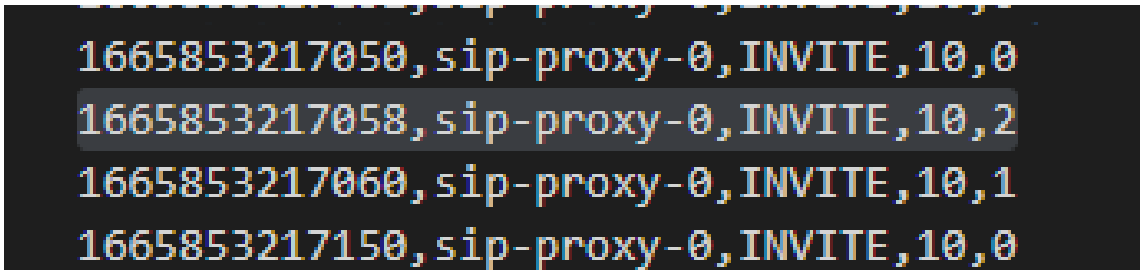


Figure 3.16: Output of the proxy analyzer

Figure 3.17: Captured packets corresponding to the figure above

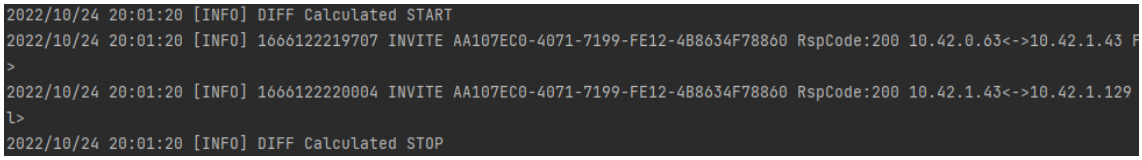


Figure 3.18: Curently calculated packet's details. 10.42.1.43 is the IP of the Proxy, which we are validating.

Chapter 4

Scaling experiments

In this chapter, our goal is to explore to possibilities of horizontally scaling the system in response to changing traffic load. We will focus on QoS requirement, which means the 99^{th} percentile of the response times (the processing time of SIP messages) should stay below a certain threshold(P_0). In our experiments, we set the threshold to $P_0 = 1000\,\text{ms}$. However, this threshold can be changed based on the request of the Operators for that particular environment.

Our environment uses the following specific configuration. The proxies run in pods within the Kubernetes cluster. As mentioned in Section 3.4, this data is scraped from the Kubernetes API. According to Kubernetes documentation [25], CPU resource is measured in CPU units. One unit equals 1 Hyperthreaded thread because we use a bare-metal Kubernetes cluster that runs on `Intel i7-10700 (16) @ 4.800GHz` with Hyperthreading enabled. Kubernetes often use mCPU or milliCPU as CPU resource notation. 1 CPU equals with 1000 mCPU and 0.1 CPU equals with 100 mCPU and so on.

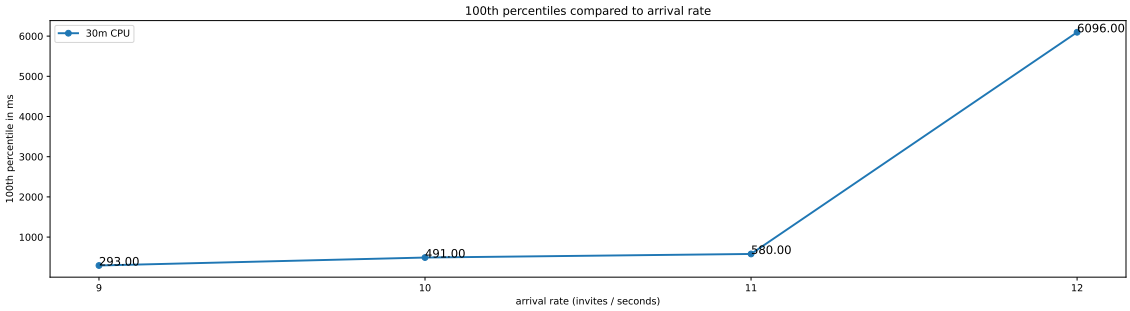
In this work, we focus on the scaling of the SIP proxy because it is likely the bottleneck component of the system. Also, the investigation by Rotter and Do [13] shows that using homogeneous components (i.e., with the same capacity) could reduce the complexity of setting scaling parameters, so we apply the SIP proxies with the same capacity in each experiment. We will perform the following steps:

- In the benchmark step at Section 4.1, we would like to know the processing capability and limit of each proxy in handling static traffic (i.e., we would like to explore how much traffic the single SIP proxy could handle to satisfy the QoS requirement).
- In Section 4.2, we will create a scenario with a deterministically changing load. We will calculate the optimal scaling decision from the processing capability of each proxy.
- In Section 4.3, we will prove our calculation by executing the scenario with the calculated fixed number of proxies. This approach is called capacity planning (i.e., we know the maximum arrival rate of the incoming traffic and prepare the system to handle that with a fixed number of components that always run)
- In Section 4.4, we will enable the built-in HPA and execute the scenario again with dynamic scaling enabled. We will try to finetune the HPA to find the most optimal performance.

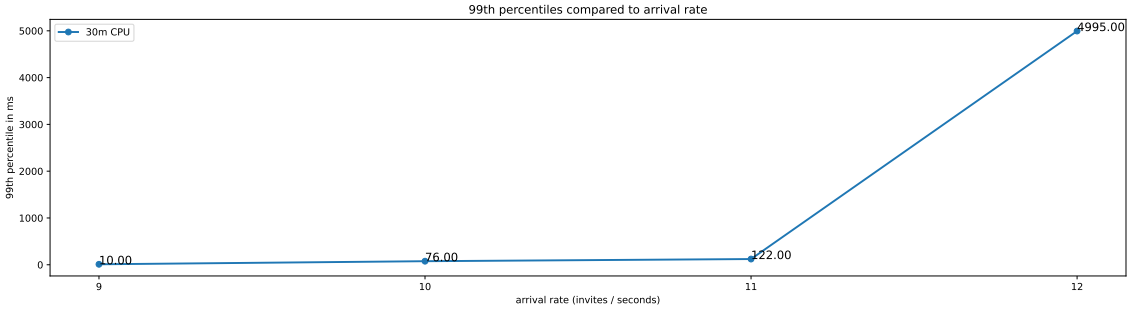
- We will execute a scenario generated from traffic shape that was collected on a real Operator's environment with real customers. For the scenario, we will use the approach that worked better.

4.1 Benchmarking

As mentioned earlier, the first step is benchmarking the proxy's behavior. We configured the CPU limit of the proxies to 30 mCPU. We created the scenario with $\lambda(t) = constant$ and set the measurement's length to 30 minutes. We are going to change the $\lambda(t)$ to find the arrival rate at which the proxy fails the QoS threshold of $P_0 = 1000\,\text{ms}$.



(a) Benchmarking: 100th percentiles of response times compared to arrival rates

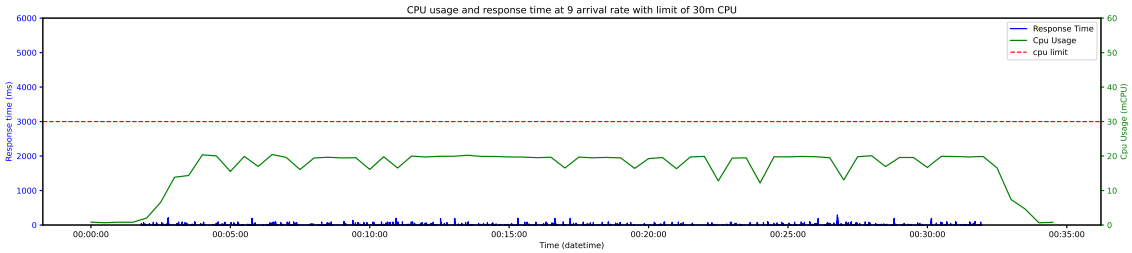


(b) Benchmarking: 99th percentiles of response times compared to arrival rates

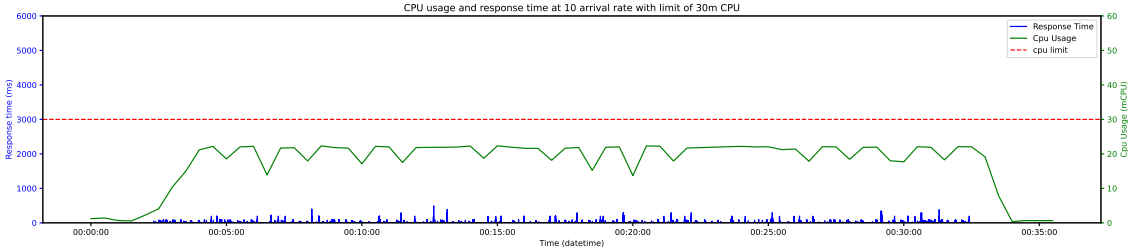
Figure 4.1: Benchmarking: Percentiles compared to arrival rate

On Figure 4.1, response times are compared to arrival rates at different CPU limits. The displayed percentile values are chosen as the following: 100^{th} percentile is equivalent to the maximum, and 99^{th} percentile is the value that comes from the definition of QoS.

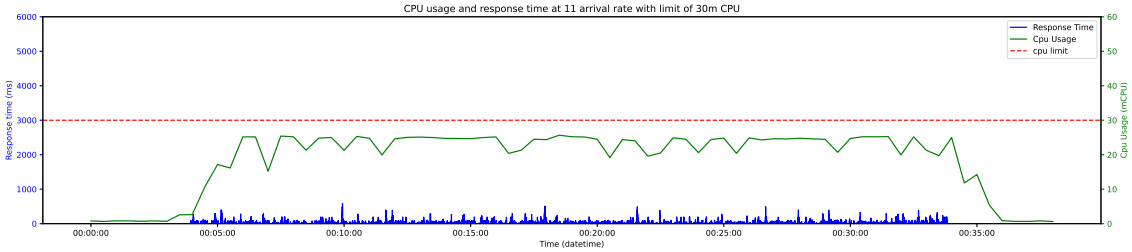
Next, compare the Figure 4.2 with the Figure 4.1. If we observe $\lambda(t)$ at 9, 10, and 11, then the percentiles of the response times are increasing but remain close to each other. When the arrival rate is 12, the QoS requirement is failed. The increasing response time was a more precise indicator of reaching the QoS threshold. The CPU usage was high between 50% to 80 %, but even the maximum response time was far less than the threshold.



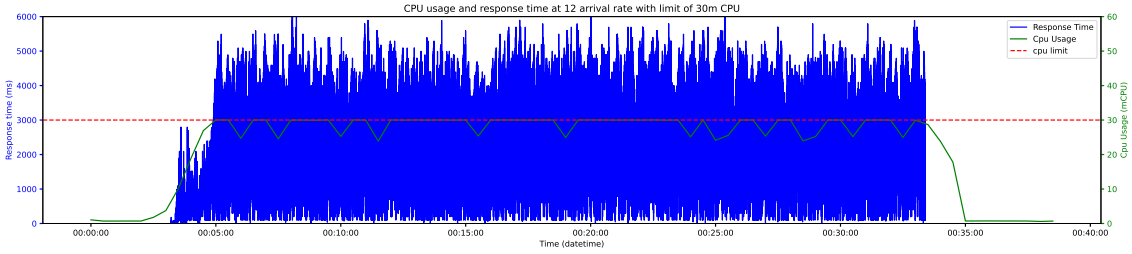
(a) Benchmarking: CPU usage and response time at $\lambda(t) = 9$ with limit of 30 miliCPU (mCPU)



(b) Benchmarking: CPU usage and response time at $\lambda(t) = 10$ with limit of 30 mCPU



(c) Benchmarking: CPU usage and response time at $\lambda(t) = 11$ with limit of 30 mCPU



(d) Benchmarking: CPU usage and response time at $\lambda(t) = 12$ with limit of 30 mCPU

Figure 4.2: Benchmarking: CPU usage and response times during a measurement with 30 mCPU

4.2 Scenario with Step funcion

The next step is to plan a scenario that forces scaling with a deterministically changing load. From Figure 4.2, we know that one proxy can handle a maximum of 11 invites/sec with the given QoS threshold. To create a small error margin, we will calculate with 10 invites/sec that a Proxy can handle maximum. That margin is required to calculate with the slight unevenness of Call-ID-based hashing. Plus, every 50 minutes, the clients must re-register, which creates an additional load.

We created a scenario where $\lambda(t)$ is a step function that can be seen on Figure 4.3. The step function starts from $\lambda(t) = 12$, and every 5 minutes, it increases with 6 invites/sec. The scenario is 30 minutes long. That six invites/sec is half the first arrival rate that one proxy can not handle.

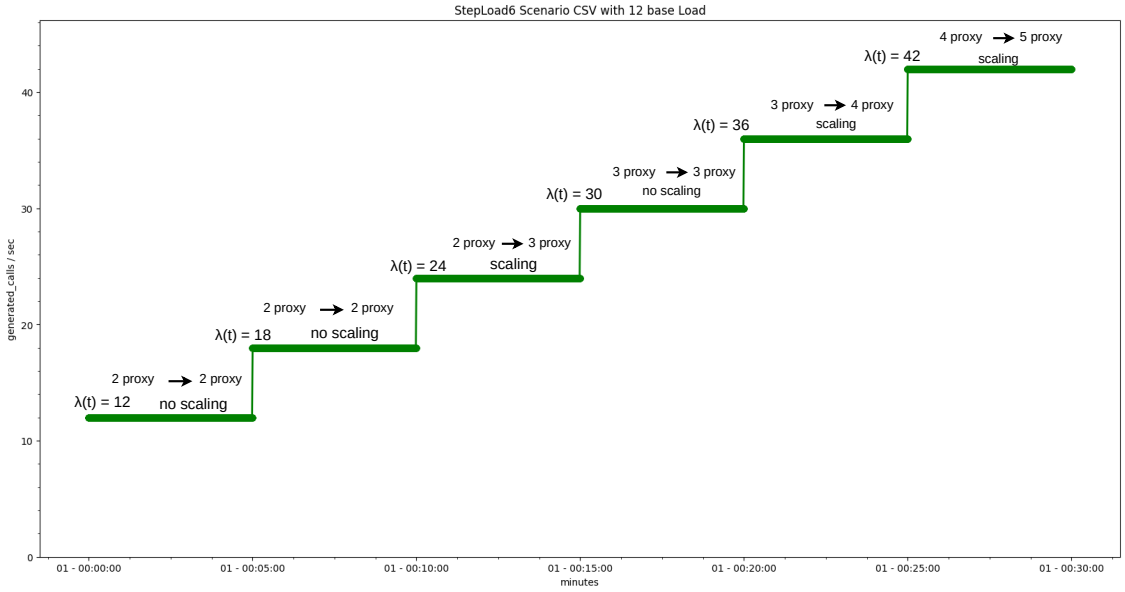


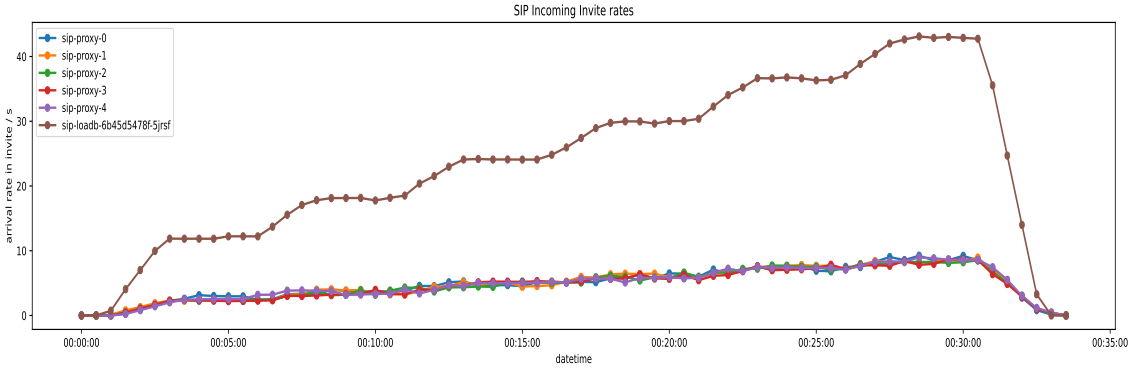
Figure 4.3: CSV of the first Scenario with Step function from $\lambda(t) = 12$ invites/sec with 6 invites/sec steps.

We also plotted the optimal scaling decision for each step on the Figure 4.3. The arrows mean the change in the number of proxies.

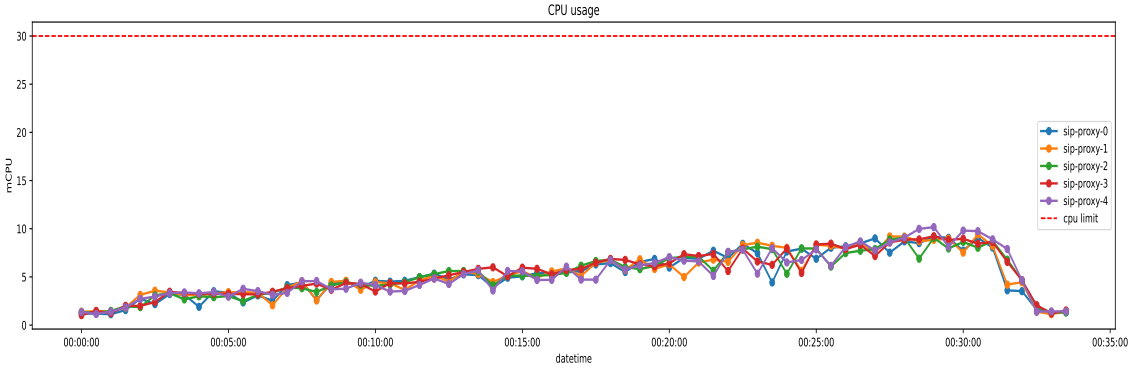
4.3 Capacity planning approach

As displayed on Figure 4.3, we planned that we would need to serve $\lambda(t) = 42$ invites/sec. If we calculate with the max 10 invites/sec per proxy, then $42/10 = 4.2$ proxies ≈ 5 proxies needed to handle the traffic of the scenario. We executed the scenario with that five proxies, and the results can be seen at Figure 4.4

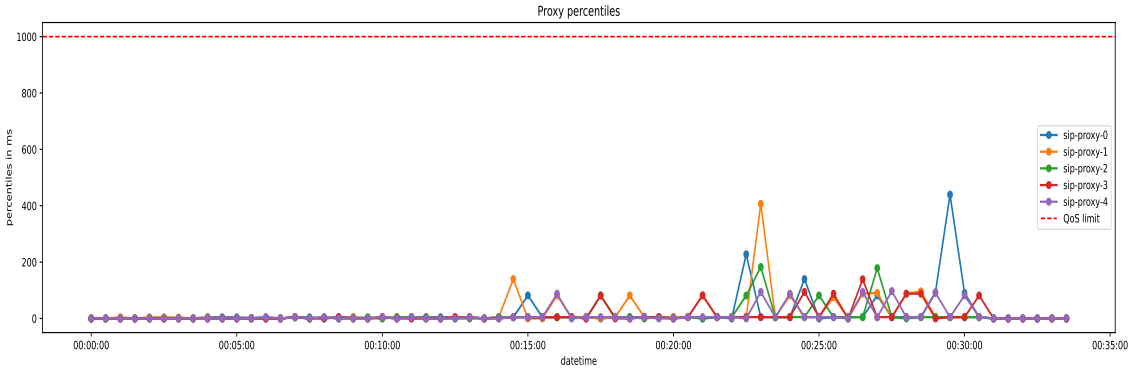
If we observe the results from Figure 4.4a, we will know that the traffic generation works as expected. Also, we can conclude that the load balancing was acceptable. From Figure 4.4b, we can see that the maximum CPU utilization was 10 mCPU. From Figure 4.4c, we can see that we were below the QoS target which means we made the QoS requirement. From Figure 4.4d, we can see that all the invites were successful.



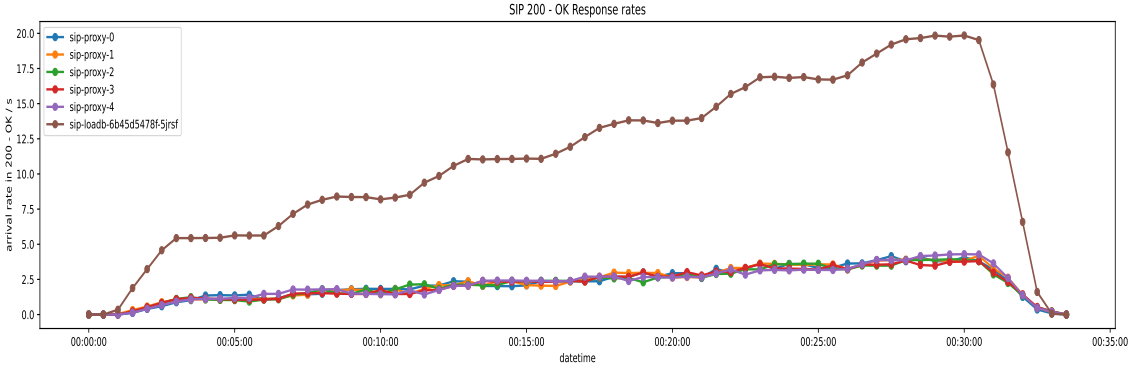
(a) SIP Incoming invite rates.



(b) CPU usage



(c) 99^{th} percentiles of response times



(d) SIP 200 - OK Response rates.

Figure 4.4: Scenario Step Load with 5 SIP Proxies

4.4 Scaling with built-in HPA

We set the HPA to scale out when the average CPU utilization across all proxies reaches 25 mCPU and stays above that for at least 10 seconds. We selected 25 mCPU first because this was the value at Figure 4.2c. We also set scale-out and scale-in maximum rates at one proxy every 60 seconds, which means the HPA must wait at least 60 seconds before adding or removing a proxy from the stateful set.

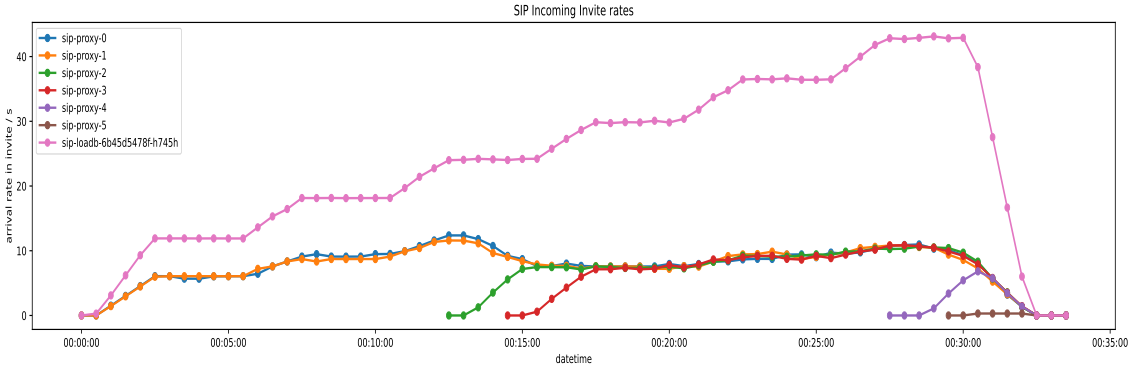
We executed the scenario, and the results are on Figure 4.5. If we observe the curve of the sip-loadb on Figure 4.5a, we will see the arrival rate of the incoming invites on the Load Balancer changes according to the step function defined previously. We can also conclude that the load balancing of the incoming request works because each proxy's arrival rate converges to the same value even after a new proxy starts. From Figure 4.5b, we can see that the 25 mCPU HPA target was too close to 30 mCPU CPU limit. The new proxy had no time to boot up and start taking some of the load. From the Figure 4.5c, we can observe that the QoS threshold was already failed when we started to reach the $\lambda(t) = 24$, which was benchmarked previously that one proxy can handle maximum $\lambda(t) = 11$. Also, we found that when the CPU usage of the proxy reaches the limit, it starts sending more 200 - OK responses, which can be seen at Figure 4.5d.

We decided to rerun the same configuration for the following scenario but change the target of the HPA to 20 mCPU. We predict this change will trigger the scaling earlier and save time for the additional proxy to start. The scenario results are displayed on Figure 4.6 in a similar format as earlier. If we look at the results, especially Figure 4.6c, we will see that there was a 5-minute time window between 00:20:00 and 00:25:00 where we again stayed below the QoS threshold after scaling. However, at 00:22:00, the HPA realized that we had reached the target of 20 mCPU, and after the 3 minutes stabilization window, it started scaling down the proxies. It has been killing five proxies in a row with the defined maximum rate of 1 proxy/60 seconds. The CPU usage started rising, and when it killed the 5^{th} proxy, it created a new one. The QoS Threshold failed again. With the previous scenario's experience, we decided to rerun the last scenario but set the target of the HPA at 10 mCPU. The 10 mCPU utilization was observable between 00:20:00 and 00:25:00.

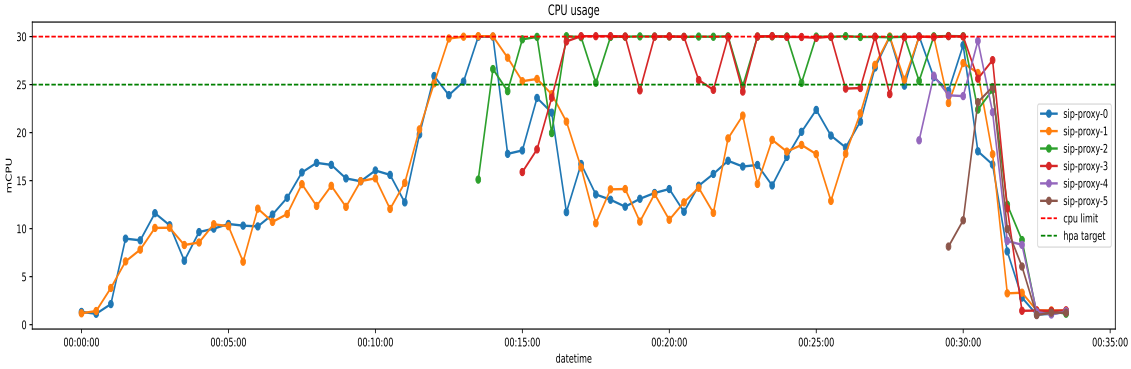
After executing the scenario, we got the results on Figure 4.7. Although we made better QoS results, we can see that we scaled too much. If we observe the CPU usage on Figure 4.7b and compare it with HPA target, we will see no matter how much we scaled we were above the target, so we scaled more. We can also see from the results that the HPA does not give enough boot time to the proxies, so it scaled up too fast. For the next experiment, we decreased the rate of the proxy adding or removing of the HPA to 1 proxy/120 s and increased the stabilization window to 30 seconds. With that changes, we believe it will further improve our results.

The results of the changes can be observed on Figure 4.8. From Figure 4.8c we improved compared to our first try on Figure 4.5c. However, on Figure 4.8d we can observe two camelback humps at 00:09:30 and 00:13:00 on the time of those humps there are percentile peaks on Figure 4.8c, and that time there were two CPU peaks on Figure 4.8b. Those details mean the first scaling was late. We tried to improve that, so we decreased the target of HPA to 5 mCPU.

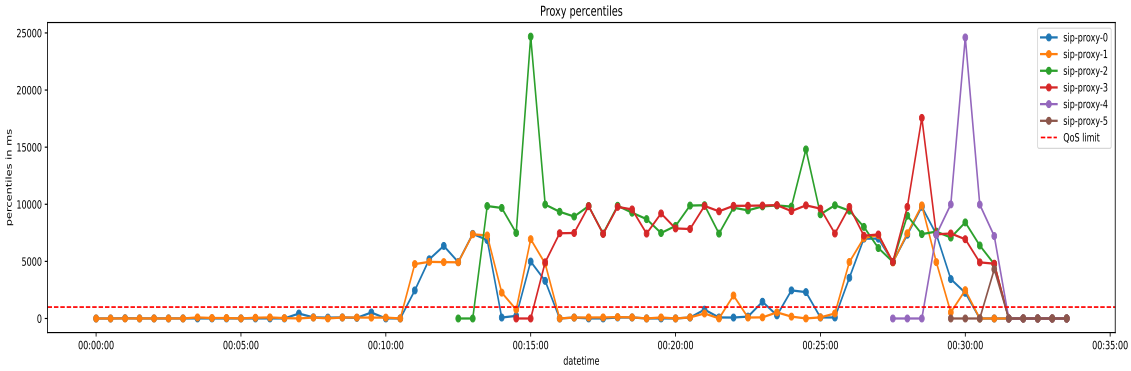
The last step scenario results can be seen on Figure 4.9. We again improved, and the camelback humps are disappeared too. However, we reached that with almost always scaling, which is not what we want. We have got the limits of our testbed. To improve



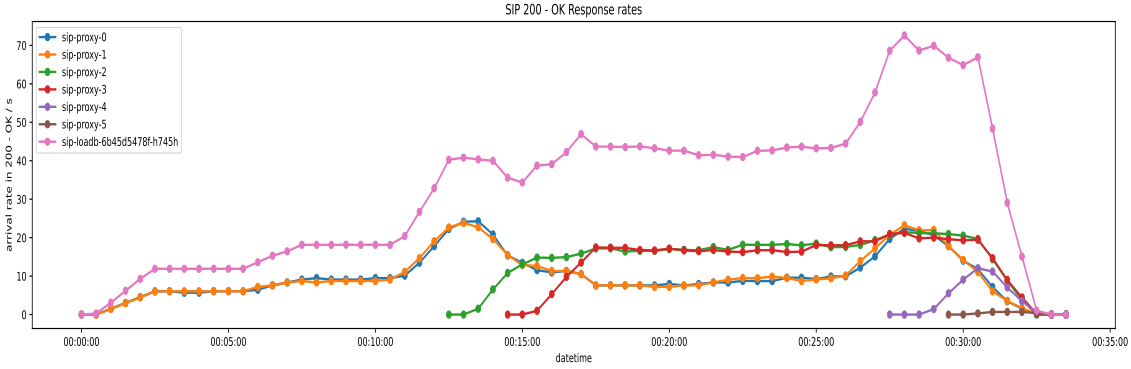
(a) SIP Incoming invite rates.



(b) CPU usage

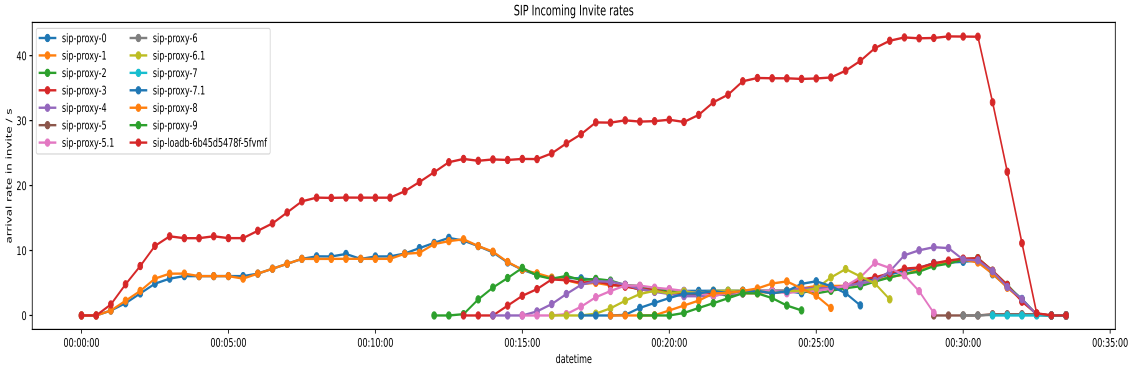


(c) 99^{th} percentiles of response times

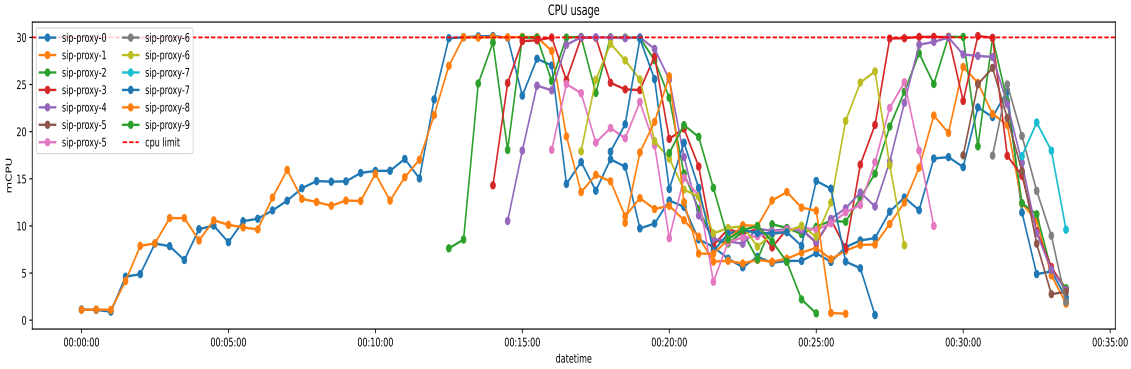


(d) SIP 200 - OK Response rates.

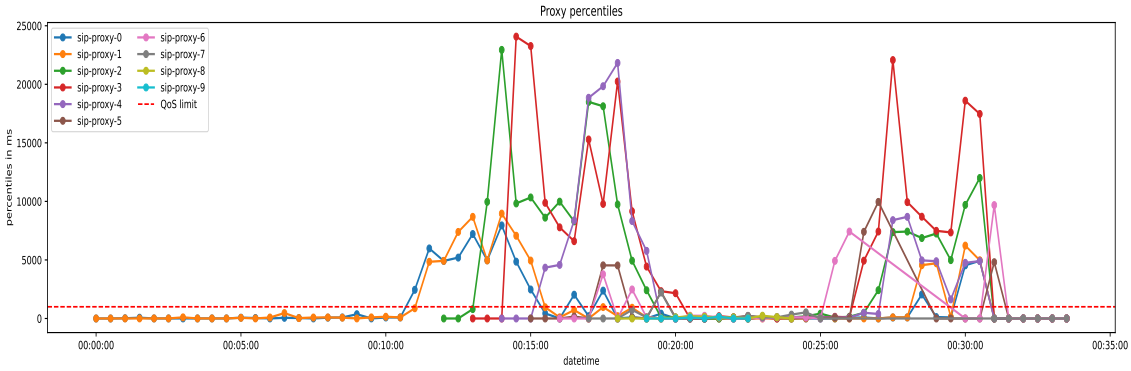
Figure 4.5: Scenario Step Load with 25 mCPU HPA target



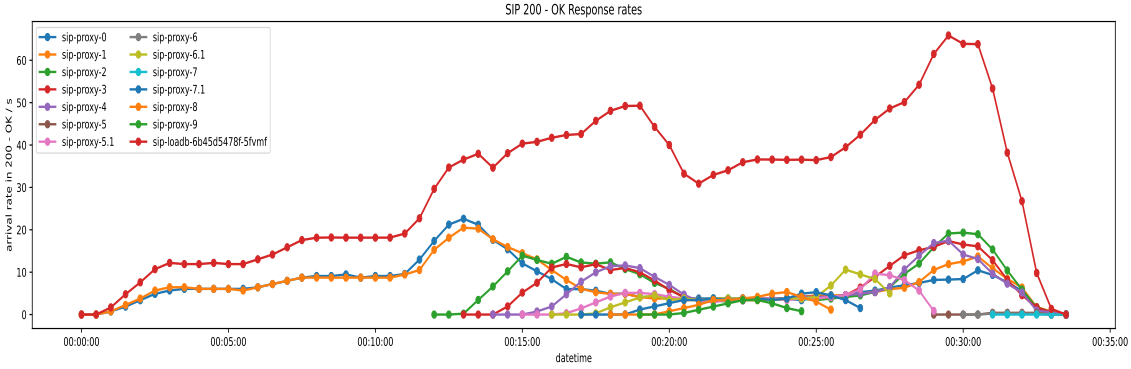
(a) SIP Incoming invite rates.



(b) CPU usage

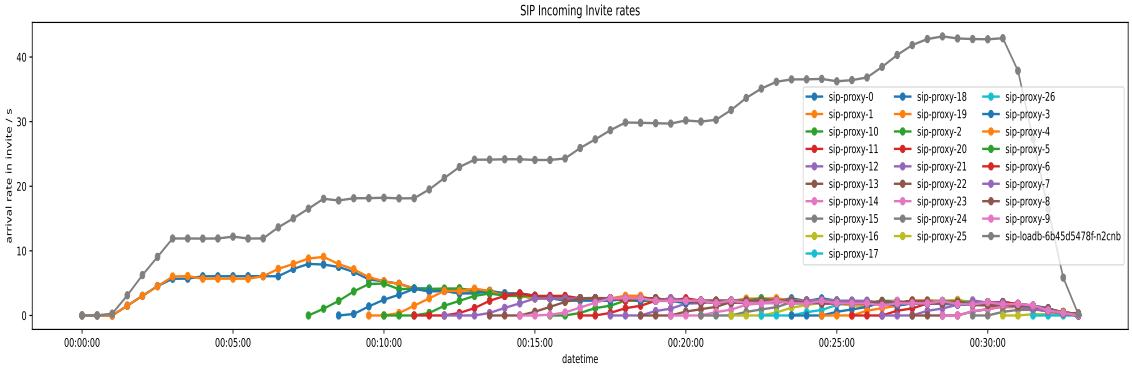


(c) 99^{th} percentiles of response times

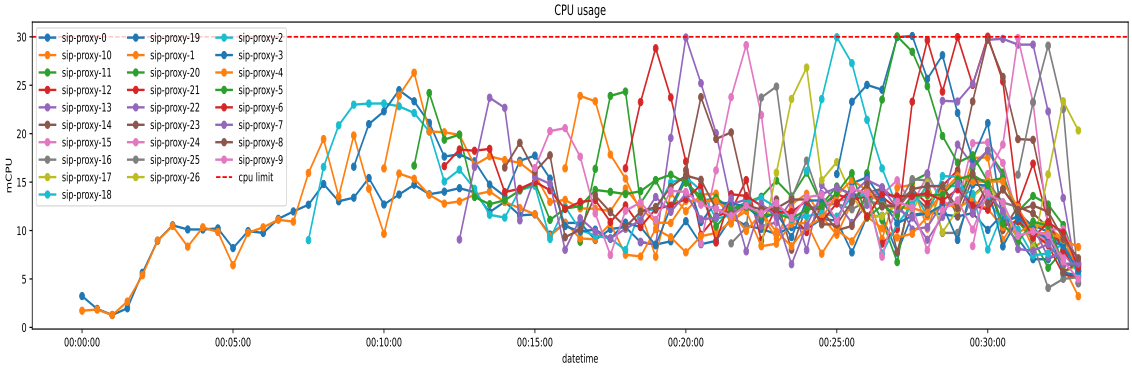


(d) SIP 200 - OK Response rates.

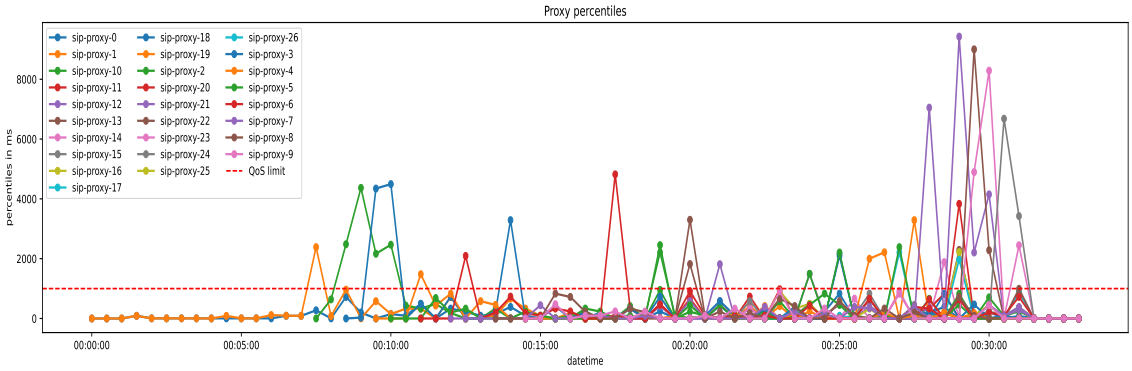
Figure 4.6: Scenario Step Load with 20 mCPU HPA target



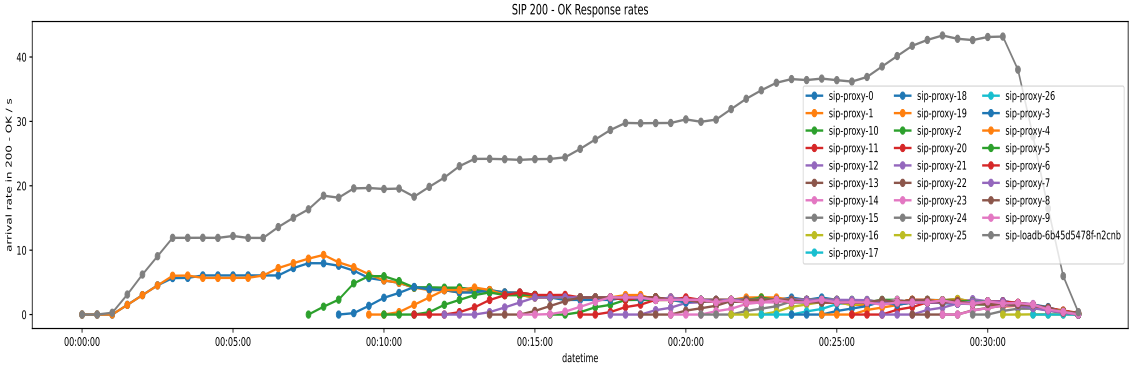
(a) SIP Incoming invite rates.



(b) CPU usage

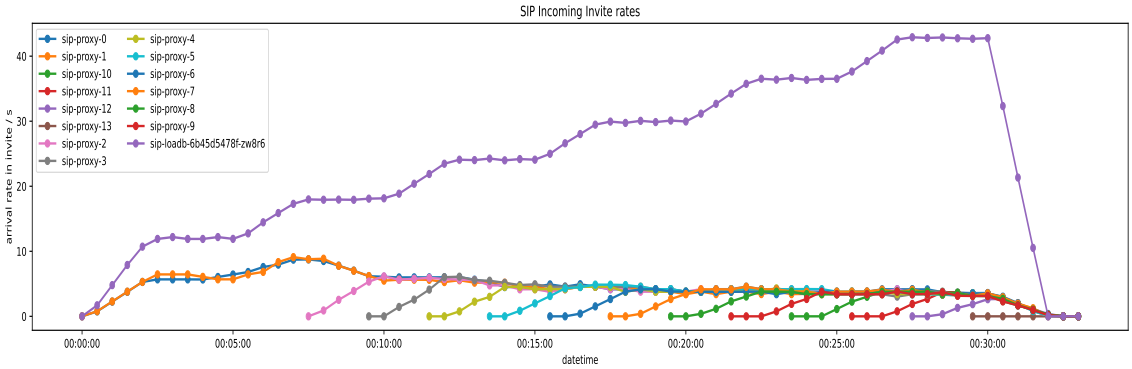


(c) 99^{th} percentiles of response times

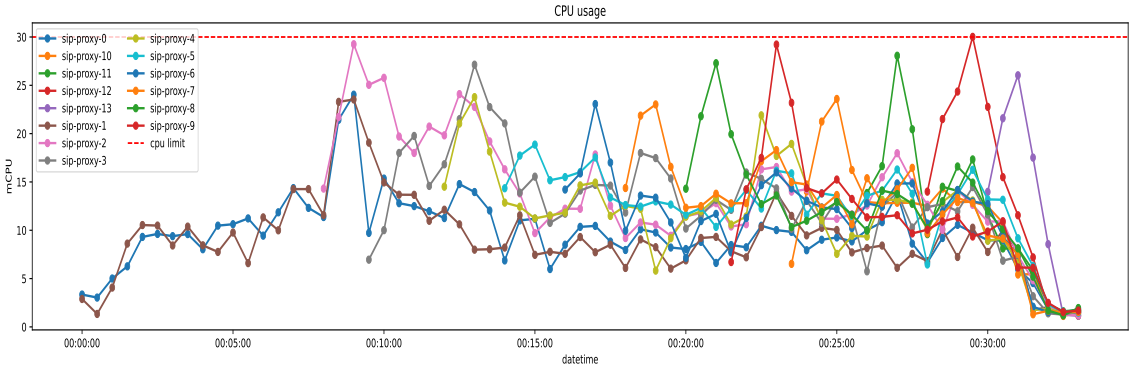


(d) SIP 200 - OK Response rates.

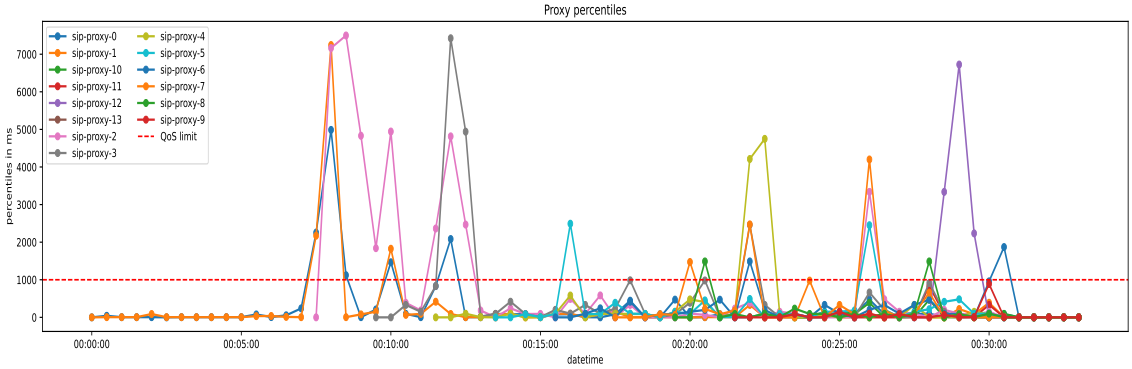
Figure 4.7: Scenario Step Load with 10 mCPU HPA target



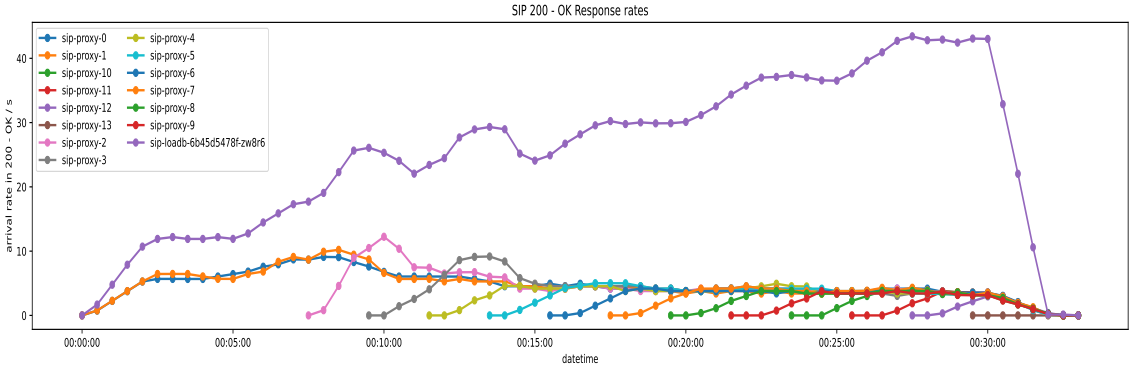
(a) SIP Incoming invite rates.



(b) CPU usage

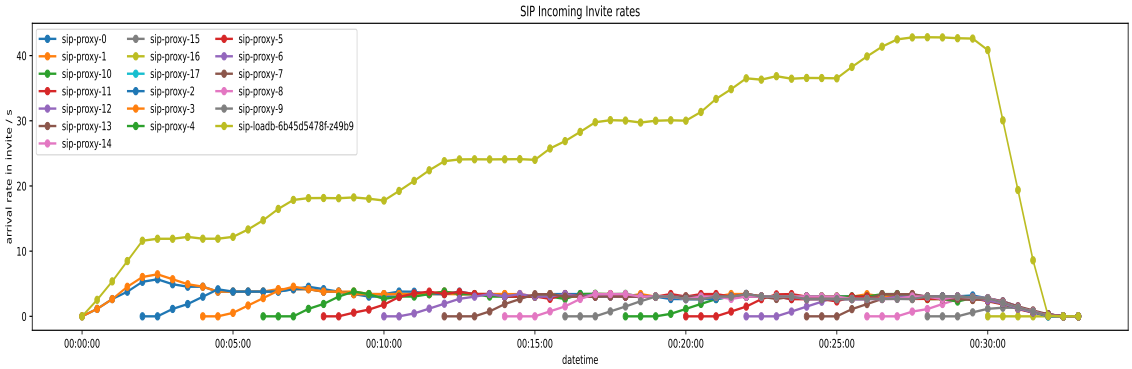


(c) 99^{th} percentiles of response times

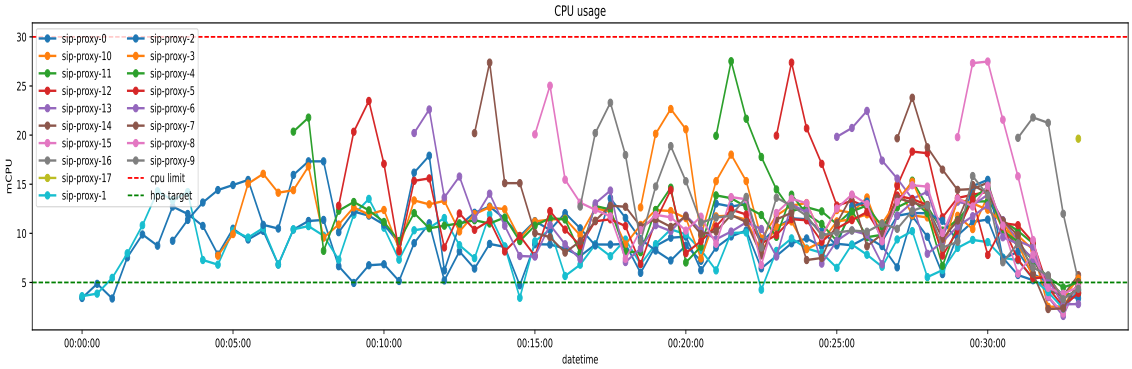


(d) SIP 200 - OK Response rates.

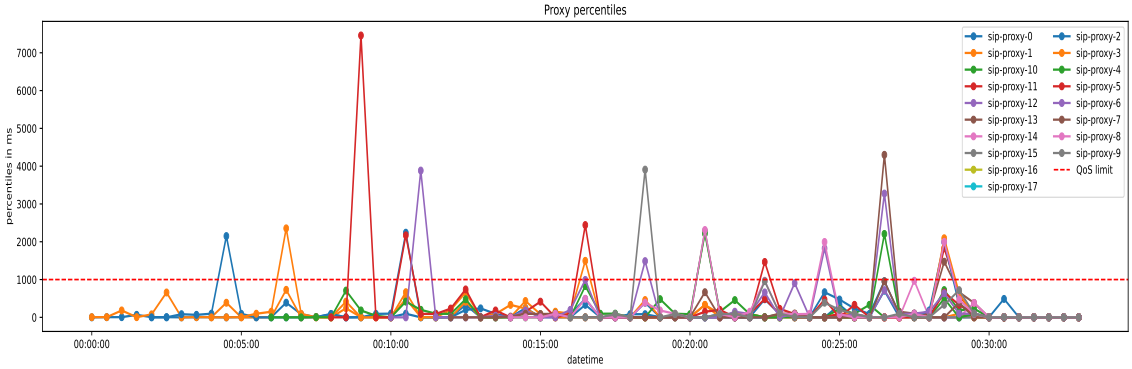
Figure 4.8: Scenario Step Load with 10 mCPU HPA target, with 1 proxy/120 s proxy changing rate and with stabilization window of 30 s



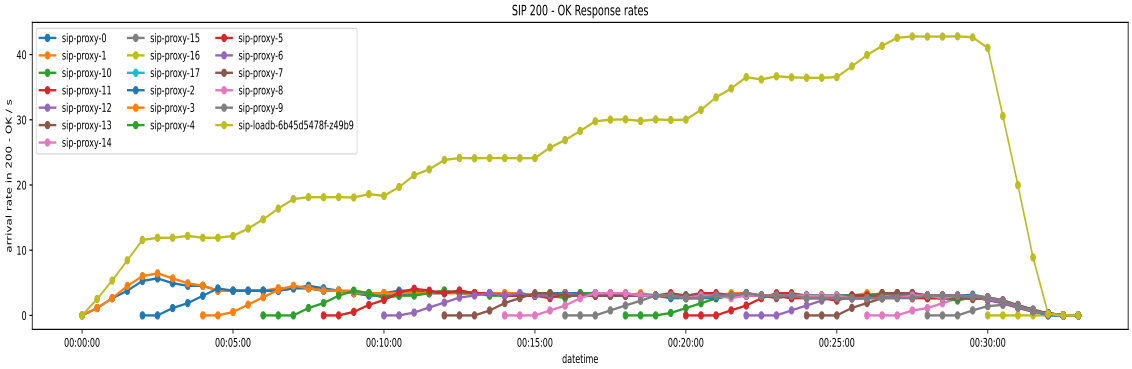
(a) SIP Incoming invite rates.



(b) CPU usage



(c) 99^{th} percentiles of response times



(d) SIP 200 - OK Response rates.

Figure 4.9: Scenario Step Load with 5 mCPU HPA target, with 1 proxy/120 s proxy changing rate and with stabilization window of 30 s

the scaling, we have to eliminate our major issue, which is the limitation of the stateful set scaling. With a stateful set, we cannot do batch scaling because a pod of the stateful set stays in a pending state until the previous stateful set pod is running. We will discuss this idea in Chapter 5. The HPA configuration with 10 mCPU target can be found at Section A.7.

We can conclude that capacity planning currently outperforms the HPA with CPU utilization-based scaling of stateful sets in our testbed. However, further testing is needed because we should also experiment with scaling of deployments. However, we have defeated a significant roadblock to converting the stateful set into deployment and factoring out the state of dialogs to fast in-memory databases like Redis [26].

4.5 Scenario with real-world traffic shape

After successfully using the capacity planning approach with the Step function scenario, we wanted to test the System with more natural traffic. However, our testbed is not mature enough to deploy and test it with customers. Our Load generator can use any $\lambda(t)$ function until it fulfills the criteria described in Section 3.3.3. We created a function that computes $\lambda(t)$ according to a dataset [27] collected from a Telecommunication (Telco) network that is used by users every day. The visualization of the calls generated from the first week of the dataset can be found at Figure 4.10. We have scaled down the $\lambda(t)$ to fit our testbed's size. We also created a shorter scenario from the first day of the dataset for initial testing.

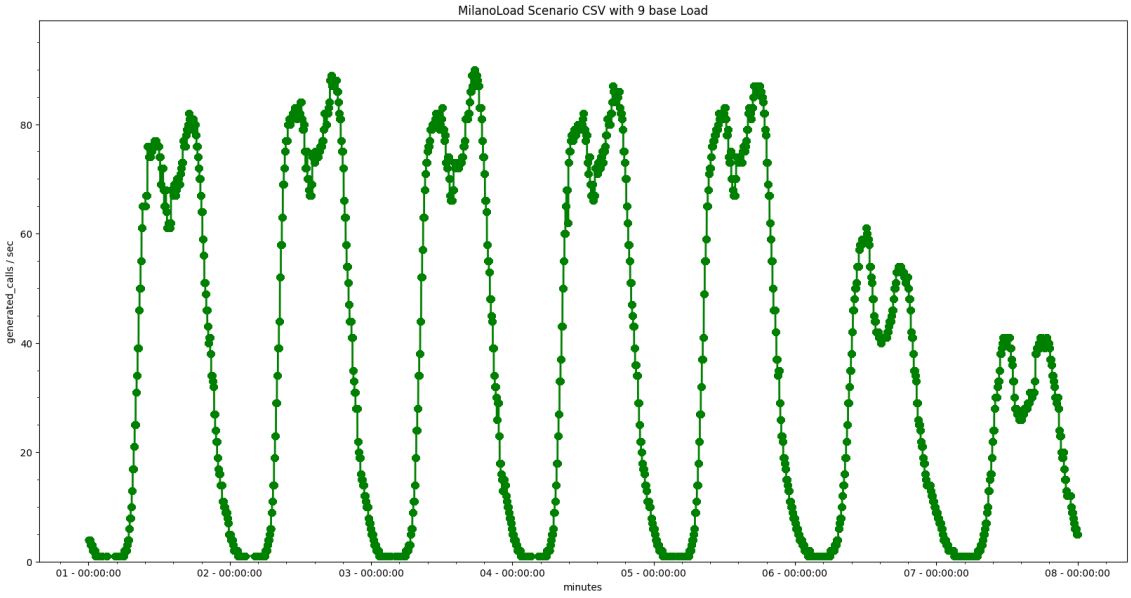


Figure 4.10: CSV of the Scenario created from Milano Dataset.

We decided to run the shorter and faster version of the dataset, displayed on Figure 4.11. We will use the capacity planning approach. We can read from the Figure 4.11 that $40 < \max \lambda(t) < 50$, so we would need five proxies to handle the scenario successfully. The results are displayed on Figure 4.12. We can conclude that we successfully executed the scenario. We almost managed to keep the QoS requirement. If we compare Figure 4.12c with Figure 4.12e we can see that 99^{th} percentiles increased at the same time when clients sent the Register requests. That requests mean an additional 2 requests/sec for each Proxy. That can easily cause a Percentile increase. Even if the registers are not counted in the percentiles, they mean an additional load for the Proxies.

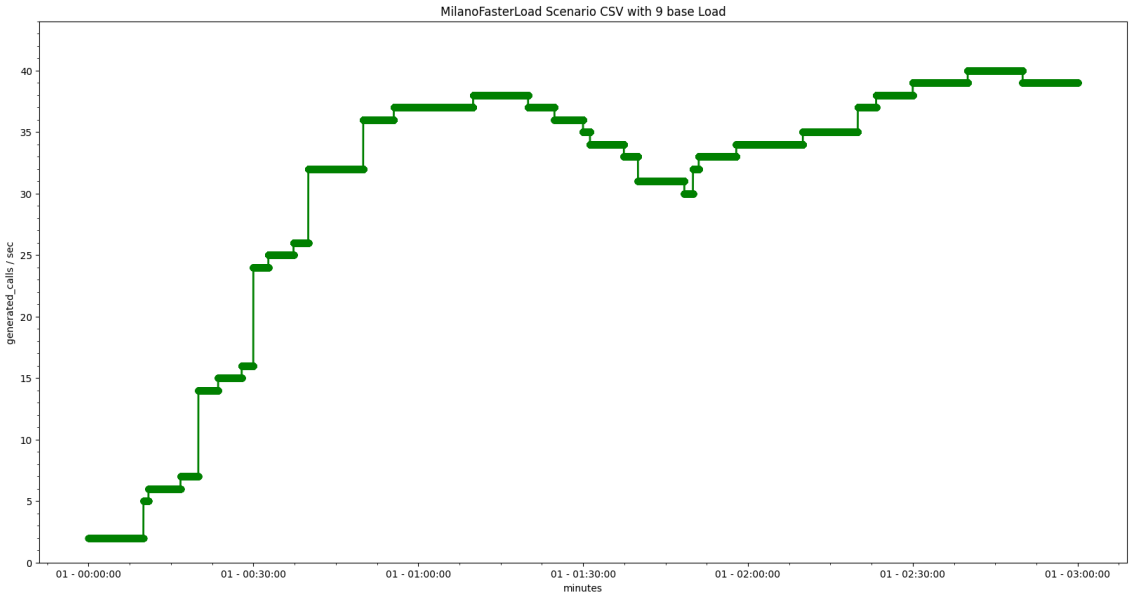
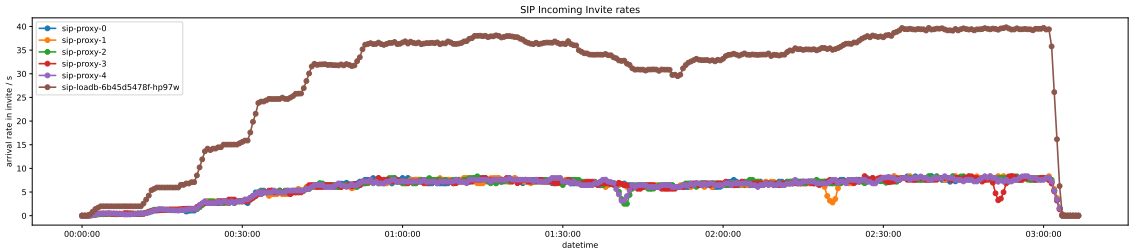
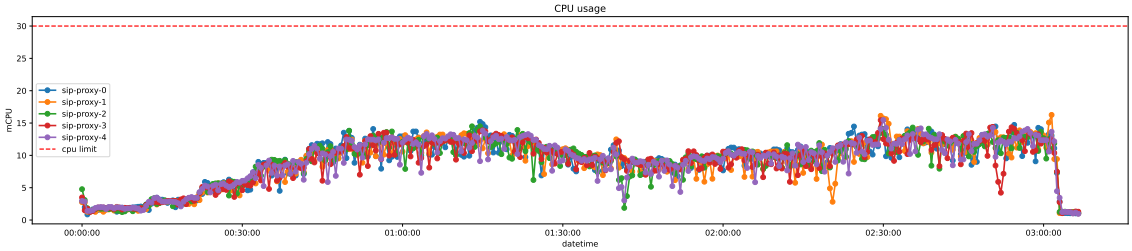


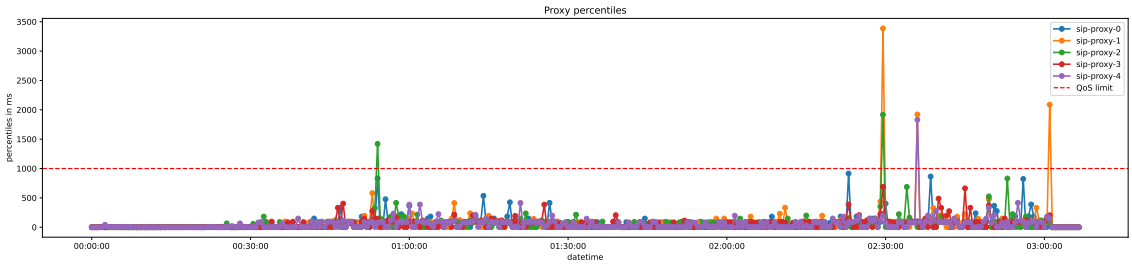
Figure 4.11: CSV of the Scenario created from Milano Dataset. 3 hours version.



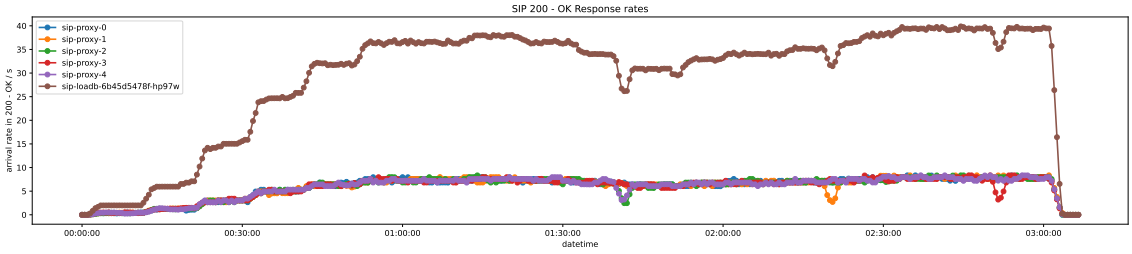
(a) SIP Incoming invite rates.



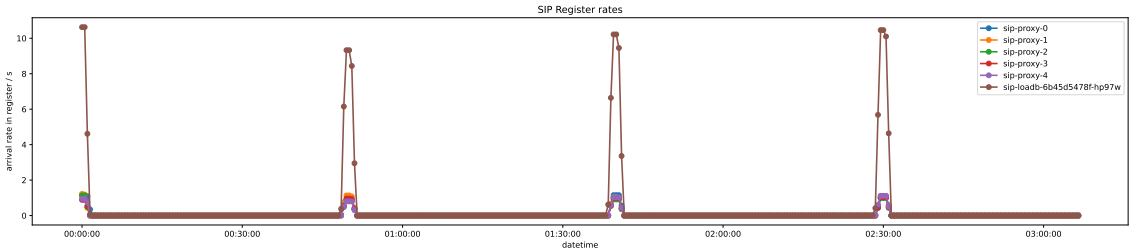
(b) CPU usage



(c) 99^{th} percentiles of response times



(d) SIP 200 - OK Response rates.



(e) SIP Register message rates.

Figure 4.12: Faster Milano Scenario with 5 proxies

Chapter 5

Summary and plans for the future

As presented in Section 2.2 we had three main goals. Firstly plan a CN IMS environment capable of executing scaling experiments. Secondly, find the limitations of CPU utilization based HPA and provide a better alternative for scaling. Thirdly, create a proof of concept for the testbed and the alternative.

We have planned and created a Kubernetes-based testbed capable of scaling experiments. The idea of separating the SIP client controller and CSV generation worked well. We also learned that testing and debugging a complex system is not straightforward. Our approach to validate the system from traffic generation to metric collection showed a few mistakes we would never find and correct with the test of each component individually. We also realized the importance of stateless applications if we want the scale the system fast and dynamically.

It is worth emphasizing that we successfully find the limitations of the HPA. We also demonstrated that CPU usage-based scaling alone is ineffective. However, with the presented approach of benchmarking, then finding the optimal HPA configuration with the guidelines of QoS parameters and validating the configured HPA with real-world traffic shape improved the HPA far more than expected. We also realized that CPU utilization is valuable information, and combining it with other metrics can cross-validate the scaling decision.

Finally, improving the testbed with a professional IMS system and the load generator is one of our future works. Also, we would like to investigate further scaling possibilities based on the combination of CPU usage and QoS parameters.

Acknowledgements

We thank our supervisors, Dr. Do Van Tien and Rotter Csaba. They guided us and gave us inside the telecommunication industry to better understand the problems that must be solved. We hope that with this TDK and our future work, we can help to solve these problems and make innovations in the future with our joint work.

We would also like to thank our friends Rafael László and Robotka Adrián for their patience and the help us to understand the world of Kubernetes deeply. They were our mentors in the early stage of our college career, and we learned much from them about Kubernetes and monitoring.

Appendix

A.1 Intro into SIP RFC

As mentioned earlier in Section 2.1, IMS participates heavily in the setup of modern phone calls. One of the most important protocols in IMS is SIP. SIP was standardized as RFC standard with the number 3261 [28]. SIP was inspired by HTTP message format, so similarities and references can be found. SIP uses a client-server architecture, meaning requests and responses are received. The definitions are cited from RFC 3261 at Table A.1 to clarify some important terms related to SIP.

Table A.1: Relevant SIP terms and its definitions

Term	Definition
Call	A call is an informal term that refers to some communication between peers, generally set up for the purposes of a multimedia conversation.
Dialog	A dialog is a peer-to-peer SIP relationship between two UAs that persists for some time. A dialog is established by SIP messages, such as a 2xx response to an INVITE request. A dialog is identified by a call identifier, a local tag, and a remote tag.
Registrar	A registrar is a server that accepts REGISTER requests and places the information it receives in those requests into the location service for the domain it handles.
Session	From the SDP specification: "A multimedia session is a set of multimedia senders and receivers and the data streams flowing from senders to receivers. A multimedia conference is an example of a multimedia session "
SIP Transaction	A SIP transaction occurs between a client and a server and comprises all messages from the first request sent from the client to the server up to a final (non-1xx) response sent from the server to the client. If the request is INVITE and the final response is a non-2xx, the transaction also includes an ACK to the response. The ACK for a 2xx response to an INVITE request is a separate transaction.
Stateful Proxy	A logical entity that maintains the client and server transaction state machines defined by this specification during the processing of a request, also known as a transaction stateful proxy.
Stateless Proxy	A logical entity that does not maintain the client or server transaction state machines defined in this specification when it processes requests. A stateless proxy forwards every request it receives downstream and every response it receives upstream.

According to RFC 3261, SIP messages use the following structure:

```
Request-Line / Status-Line
*message-header
CRLF
[ message-body ]
```

In the case of a request, the first line contains the following in order, separated by whitespace: a SIP Method, the Request-URI, and the SIP version. The next one or more lines are the message headers, each at a new line. After that, a mandatory Carrige Return Line Feed (CRLF) comes. From that until the end of the message, the optional message body takes place. From the defined six SIP Methods, the following is important to us: REGISTER to register contact information, INVITE and ACK to setup up sessions, and OPTIONS for querying server for capabilities. The Request-URI is a SIP or SIP Secure (SIPS) Uniform Resource Identifier (URI) that identifies communication resources, e.g., `sip:0000@example-sip.svc.cluster.local` where `sip` means the SIP protocol, `0000` is the prefix of the client and `example-sip.svc.cluster.local` is the suffix. SIP version is `SIP/2.0` in our context.

In the case of a response, the structure of the first line changes. In the case of a response, the first line contains the following in order, separated by whitespace: a SIP version, a status code, and a reason phrase. The version is the same as presented previously. The status code is a 3-digit integer to indicate the result of the request. The reason phrase is similar to the status code but for human beings. The status codes used in our paper are listed at Table A.2.

Table A.2: Relevant SIP status codes

Status Code	Reason Phrase	Description
100	Trying	request received by the next-hop-server, processing started
180	Ringing	UA received the INVITE
200	OK	The request was successful
401	Unauthorized	The request requires authentication, the Registrar sends it
404	Not Found	The server knows for sure that the requested user does not exist at the domain
407	Proxy Authentication Required	The request requires authentication, the Proxy sends it
483	Too Many Hops	Max forward field is zero, which means the max hop count reached

As mentioned earlier, SIP uses a client-server model. Participants in a SIP call are the following: SIP Clients on both ends of the communication, stateful or stateless proxies, and registrars. Those components (except for the stateless proxy) have some standard core functionality. Depending on the situation, those cores can act as an User Agent Client (UAC) or as an User Agent Server (UAS). For example, Client - A originates a call so that it will act as an UAC. Proxy - 0 will accept that request acting as an UAS and forward that call statefully acting as an UAC to Client - B. Client-B handles that request acting as an UAS. From the previous example, we can conclude that UACs generates requests, and UASs handles that request and sends a response back. A component can also act as an UAC and as an UAS in the same call but a different transaction.

When UAC generates a request, the SIP message header must contain the following fields: `To`, `From`, `CSeq`, `Call-ID`, `Max-Forwards`, and `Via`. The `To` field contains information about the logical destination of the message. The `From` contains information about the message's originator. `CSeq` is a sequence number to identify and order transactions. `Call-ID` is an identifier to group messages together in a dialog. `Max-Forwards` serves as a limit of hops that a message can travel. Similar to Time-to-live (TTL) in the context of Internet Control Message Protocol (ICMP). The `Via` header identifies where a request-response is to be sent. Multiple `Via` headers are possible in a message, e.g., an Invite arrives at SIP Client - B from the previous example, then it contains two `Via`s in the following order. The topmost `Via` will contain the address of Proxy - 0, and the next `Via` will contain the address of Client - A. When the response is sent back to Proxy - 0, it will remove the topmost `Via` and forward the response to Client - A. The order of the `Via`s works similarly to stack memory. Put the new `Via` to the top of the stack. Remove and process the old `Via` from the top.

INVITEs originated from SIP Clients acting as a UAC and terminated at another SIP Clients acting as a UAS. Clients can move, and with that, there is no guarantee to keep their original IP addresses. However, in SIP, there is a built-in solution for the creation of bindings between IP addresses and SIP URIs, which are bound to a particular user. This process is called registration. During registration, a SIP client sends a REGISTER request to a specific UAS called Registrar. If the Registrar, after the authentication and authorization of the SIP Client, accepts the request, then an AoR is created in its DB. This AoR contains contact information in which the IP address of the client is stored. AoR has an expiration time of often 3600 seconds. In that period, the client can re-register to refresh that value. After expiration, the AoR record is removed from the DB of the Registrar. After a successful Registration request, the Registrar sends back a response to the Client with a status of `200 OK` as seen on Figure A.1.1 and Registration at Section 3.2.1.

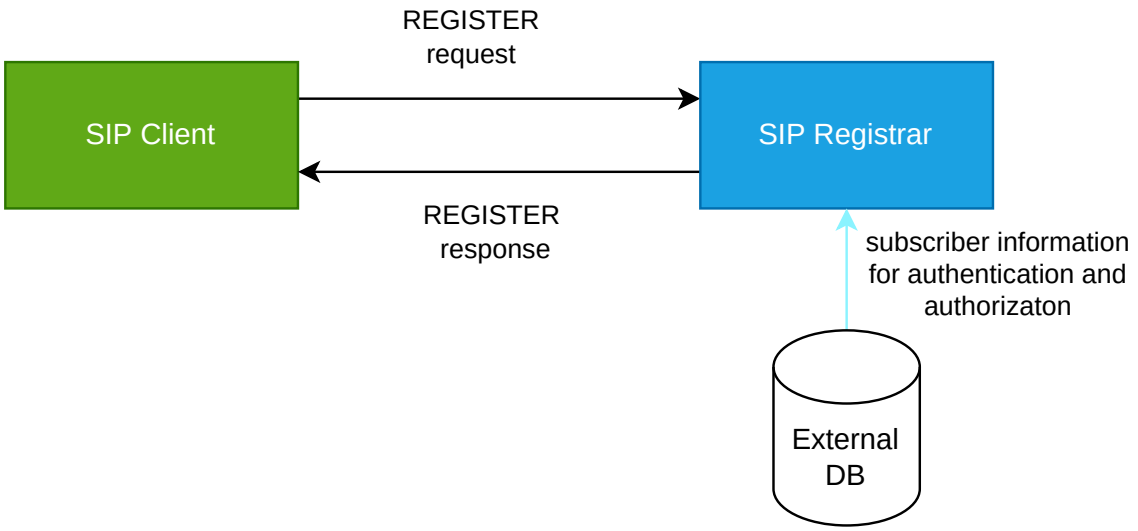


Figure A.1.1: SIP Register

In our simplified CN IMS, a stateful SIP server will act as a Registrar called SIP Proxy. This component is also responsible for forwarding SIP INVITEs to the called SIP client. We decided to merge those components because the number of registrations compared to the number of invites is negligible. However, it significantly decreased the complexity of the implementation of the system. In our example, SIP Client - A sends an INVITE to the SIP Proxy. After successful authentication and authorization, the request is forwarded

to the terminating SIP Client, in this example, to SIP Client - B. This Client process the request and answers with a response. This response is sent to the SIP Proxy. Then the response is forwarded back to the originator, in our example, to SIP Client - A as seen on Figure A.1.2.

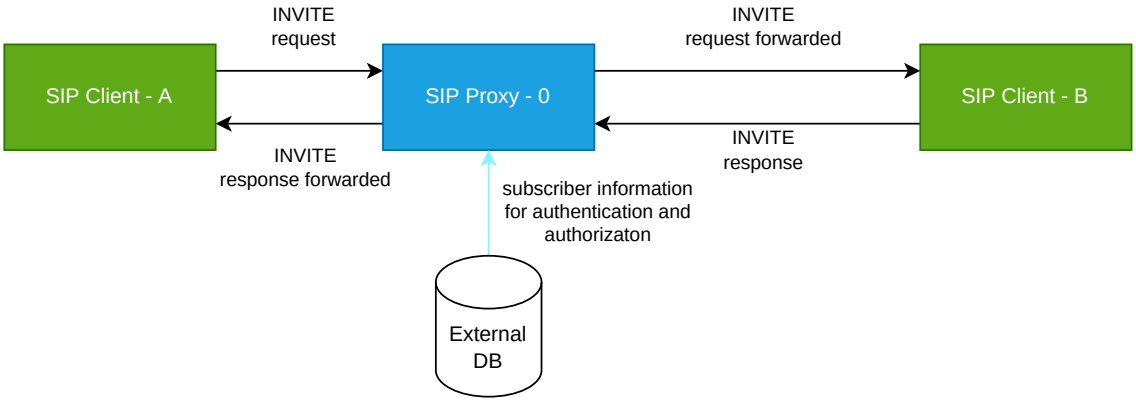


Figure A.1.2: SIP Invite

A.2 Horizontal scaling compared to Vertical scaling

Kubernetes can increase the available resources for the deployed applications horizontally or vertically to help the Operators defeat the challenges successfully. Vertical scaling means adding more power to the pods, which means increasing the defined CPU or upper memory bounds of the pod. Horizontal scaling means increasing the number of pods for the same purpose. For vertical scaling, the hard upper limit is the hardware on which the pods run. Horizontal scaling seems a better approach because of the hardware limitation and because the Kubernetes system was also developed with flexible node management(adding or removing physical nodes to the cluster).

A.3 Kamailio Dockerfile

```
FROM kamailio/kamailio:5.5.0-xenial
RUN apt update && export DEBIAN_FRONTEND=noninteractive && apt install -yq \
    netcat iproute2 net-tools netcat tcpdump iputils-ping host curl \
    rsyslog \
    && rm -rf /var/lib/apt/lists/*
RUN ln -s /usr/sbin/tcpdump /tmp/static-tcpdump

COPY entrypoint.sh /entrypoint.sh

COPY defaults/ /etc/kamailio/

EXPOSE 5060
ENTRYPOINT ["/entrypoint.sh"]
```

A.4 Kamailio entrypoint.sh

```
#!/bin/bash

sed -i "s/example-sip.svc.cluster.local/$SIP_DOMAIN/g" /etc/kamailio/kamailio.cfg
sed -i "s/SERVER_ADDRESS_PLACEHOLDER/$(hostname -I|xargs)/g" /etc/kamailio/kamailio.cfg
```

```
kamctl address reload
kamailio -DD -E -e
bash -c "tail -f /dev/null"
```

A.5 SIP-proxy config

```
/* Main SIP request routing logic
* - processing of any incoming SIP request starts with this route
* - note: this is the same as route { ... } */
request_route {

  route(HANDLE_DMQ);

  route(REQINIT);

  if(method=="OPTIONS"){
    sl_reply("200","OK");
    exit;
  }

  if (!is_method("ACK")) {
    if(t_precheck_trans()) {
      t_check_trans();
      exit;
    }
    t_check_trans();
  }

  # handle requests within SIP dialogs
  route(WITHINDLG);

  if(method=="INVITE"){
    #xlog("meme: with auth source: $rm");
    if (!auth_check("$fd", "subscriber", "0")) {
      auth_challenge("$fd", "0");
      exit;
    }
    route(ONNETINVITE);
  }

  if(method=="REGISTER"){
    if (!auth_check("$fd", "subscriber", "0")) {
      auth_challenge("$fd", "0");
      exit;
    }
    save("sip-loadb.example-sip.svc.cluster.local"); #http://www.kamailio.org/docs/modules/5.0.x/
     modules/registrar.html see 4.1  save(domain, [, flags [, uri]])
    #!ifdef WITH_DMQ
    dmq_t_replicate();
    #!endif
    exit;
  }
  xlog("No idea how to respond to method $rm \n");
  sl_reply("501", "Not Implemented");
}

route[HANDLE_DMQ]{
  if(is_method("KDMQ")){
    dmq_handle_message();
  }
}

route[ONNETINVITE]{
  if(!lookup("sip-loadb.example-sip.svc.cluster.local")){
    sl_reply("404", "User not Registered");
    exit;
  }
```

```
  lookup("sip-loadb.example-sip.svc.cluster.local");
  t_relay();
  exit();
}

# Per SIP request initial checks
route[REQINIT] {
  # no connect for sending replies
  set_reply_no_connect();
  # enforce symmetric signaling
  # - send back replies to the source address of request
  force_rport();

  if (!mf_process_maxfwd_header("10")) {
    sl_send_reply("483","Too Many Hops");
    exit;
  }

  if(is_method("OPTIONS") && uri==myself && $rU==$null) {
    sl_send_reply("200","Keepalive");
    exit;
  }

  if(!sanity_check("17895", "7")) {
    xlog("Malformed SIP request from $si:$sp\n");
    exit;
  }
}

# Handle requests within SIP dialogs
route[WITHINDLG] {
  if (!has_totag()) return;

  if ( is_method("ACK") ) {
    if ( t_check_trans() ) {
      # no loose-route, but stateful ACK;
      # must be an ACK after a 487
      # or e.g. 404 from upstream server
      if (!t_relay()) {
        sl_reply_error();
      }
      exit;
    } else {
      # ACK without matching transaction ... ignore and discard
      exit;
    }
  }
  #xlog("We should follow the RFC....\n");
  sl_send_reply("404","Not here");
  exit;
}

#!ifdef WITH_XHTTP
event_route[xhttp:request] {
  xlog("L_DBG","Recieved HTTP request with request $hu\n");
  xhttp_reply("200", "OK", "application/json", "ok");
}
#!endif
```

A.6 SIP-proxy Stateful Set

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sip-proxy
spec:
  replicas: {{.Values.Containers.SipProxy.replicas}}
  selector:
    matchLabels:
```

```
    app: sip-proxy
serviceName: sip-proxy
template:
  metadata:
    labels:
      app: sip-proxy
      project: qos-based-scaling
      release: prom-stack
  spec:
  {{ if .Values.HitCluster }}
    nodeSelector:
      node: sip-node-02
  {{ end }}
    initContainers:
      - name: init-sip-proxy
        image: harbor.sch.bme.hu/private-woranhun/kamailio:{{.Values.Containers.SipProxy.version}}
        command:
          - bash
          - '-c'
          - |
            cp /tmp/etc/kamailio/* /etc/kamailio/ &&
            chmod +x /etc/kamailio/*.sh &&
            (until nc -z mysql-0.mysql 3306 > /dev/null; do echo Waiting for MYSQL...; sleep 2;
   done;) &&
            if [ "$HOSTNAME" = "sip-proxy-0" ]; then /etc/kamailio/db-init.sh; fi
        env:
          - name: SIP_DOMAIN
            value: {{ .Release.Namespace }}.svc.cluster.local
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        imagePullPolicy: Always
        volumeMounts:
          - name: sip-proxy-config
            mountPath: /tmp/etc/kamailio
          - name: sip-proxy-config-rw
            mountPath: /etc/kamailio
    containers:
      - name: sip-proxy
        image: harbor.sch.bme.hu/private-woranhun/kamailio:{{.Values.Containers.SipProxy.version}}
        ports:
          - containerPort: 5060
            protocol: UDP
          - containerPort: 8080
            protocol: TCP
        env:
          - name: SIP_DOMAIN
            value: {{ .Release.Namespace }}.svc.cluster.local
        resources:
          limits:
            cpu: 30m
            memory: 100Mi
          requests:
            cpu: 20m
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        imagePullPolicy: Always
        volumeMounts:
          - name: sip-proxy-config-rw
            mountPath: /etc/kamailio
        lifecycle:
          postStart:
            exec:
              command:
                [
                  "/bin/sh",
                  "-c",
                  "/etc/kamailio/start.sh"
                ]
          preStop:
            exec:
              command:
```

```
                [
                  "/bin/sh",
                  "-c",
                  "/etc/kamailio/stop.sh"
                ]
          readinessProbe:
            httpGet:
              path: /health
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
            timeoutSeconds: 5
        - name: sip-proxy-analyzer
          image: harbor.sch.bme.hu/private-woranhun/kamailio-analyzer:{{.Values.Containers.
    SipProxyAnalyzer.version}}
          ports:
          - name: analyzer
            containerPort: 9101
            protocol: TCP
          - name: benchmark
            containerPort: 9102
            protocol: TCP
          resources:
            limits:
              cpu: 500m
              memory: 500Mi
            requests:
              cpu: 100m
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: Always
          readinessProbe:
            httpGet:
              path: /health
              port: 8080
            initialDelaySeconds: 5
            periodSeconds: 5
            timeoutSeconds: 5
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: harbor-woranhun-private-readonly
        - name: harbor-dockerhub-proxy-readonly
      schedulerName: default-scheduler
      volumes:
      - name: sip-proxy-config-rw
        emptyDir: {}
      - name: sip-proxy-config
        configMap:
          name: sip-proxy-configmap
```

A.7 SIP-proxy HPA

```
  apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sip-proxy-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: StatefulSet
    name: sip-proxy
  minReplicas: 2
  maxReplicas: 50
  metrics:
    - type: Pods
      pods:
```

```
      metric:
        name: container_cpu_usage_per_second
      target:
        type: AverageValue
        averageValue: 10m
 behavior:
   scaleUp:
     stabilizationWindowSeconds: 30
     policies:
       - type: Pods
         value: 1
         periodSeconds: 120
     selectPolicy: Max
   scaleDown:
     stabilizationWindowSeconds: 30
     policies:
       - type: Pods
         value: 1
         periodSeconds: 120
     selectPolicy: Max
```

A.8 SIP Load Balancer Kamailio Config

```
 /* Main SIP request routing logic
 * - processing of any incoming SIP request starts with this route
 * - note: this is the same as route { ... } */
request_route {

  # per request initial checks
  route(REQINIT);

  # hash over call-id dispatching on gateways group '1'
  if(!ds_select_dst(1, 0))
  {
    send_reply("404", "No destination");
    exit;
  }
  xlog("L_DBG", "--- SCRIPT: going to <$ru> via <$du>\n");
  t_on_failure("DISPATCH_FAILURE");
  forward();

}

# Per SIP request initial checks
route[REQINIT] {
  if (!mf_process_maxfwd_header("10")) {
    sl_send_reply("483","Too Many Hops");
    exit;
  }

}

route[DISPATCH_FAILURE]{
  sl_reply("500", "Failed to relay request");
  xlog("DISPATCH_FAILURE: $rm $ci - $fu <-> $du  $mb\n");
  exit;
}

#!ifdef WITH_XHTTP
event_route[xhttp:request] {
  xlog("L_DBG","Recieved HTTP request with request $hu\n");
   xhttp_reply("200", "OK", "text/html", "<html><body>OK</body></html>");
}
#!endif
```

A.9 SIP-loadb Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sip-loadb
spec:
  replicas: {{.Values.Containers.SipLoadB.replicas}}
  selector:
    matchLabels:
      app: sip-loadb
  template:
    metadata:
      labels:
        app: sip-loadb
        release: prom-stack
        project: qos-based-scaling
    spec:
    {{ if .Values.HitCluster }}
      nodeSelector:
        node: sip-node-02
    {{ end }}
      initContainers:
        - name: init-sip-loadb
          image: harbor.sch.bme.hu/private-woranhun/kamailio:{{.Values.Containers.SipLoadB.version
    }}
          command:
            - bash
            - '-c'
            - |
              cp /tmp/etc/kamilio/* /etc/kamailio/ &&
              until nc -z mysql-0.mysql 3306 > /dev/null; do echo Waiting for MYSQL...; sleep 2;
    done;
          env:
            - name: SIP_DOMAIN
              value: {{ .Release.Namespace }}.svc.cluster.local
          resources: {}
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: Always
          volumeMounts:
            - name: sip-loadb-config
              mountPath: /tmp/etc/kamilio
            - name: sip-loadb-config-rw
              mountPath: /etc/kamailio
      containers:
        - name: sip-loadb
          image: harbor.sch.bme.hu/private-woranhun/kamailio:{{.Values.Containers.SipLoadB.version
    }}
          ports:
            - containerPort: 5060
              protocol: UDP
          env:
            - name: SIP_DOMAIN
              value: {{ .Release.Namespace }}.svc.cluster.local
          resources:
            limits:
              cpu: 600m
              memory: 300Mi
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: Always
          volumeMounts:
            - name: sip-loadb-config-rw
              mountPath: /etc/kamailio
        - name: sip-loadb-analyzer
          image: harbor.sch.bme.hu/private-woranhun/kamailio-analyzer:{{.Values.Containers.
    SipLoadBAnalyzer.version}}
          ports:
          - name: analyzer
            containerPort: 9101
            protocol: TCP
```

```
          - name: benchmark
            containerPort: 9102
            protocol: TCP
          resources:
            limits:
              cpu: 500m
              memory: 4Gi
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          imagePullPolicy: Always
      restartPolicy: Always
      terminationGracePeriodSeconds: 30
      dnsPolicy: ClusterFirst
      securityContext: {}
      imagePullSecrets:
        - name: harbor-woranhun-private-readonly
        - name: harbor-dockerhub-proxy-readonly
      schedulerName: default-scheduler
      volumes:
      - name: sip-loadb-config-rw
        emptyDir: {}
      - name: sip-loadb-config
        configMap:
          name: sip-loadb-configmap
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
  revisionHistoryLimit: 10
  progressDeadlineSeconds: 600
```

A.10 SIP-loadb Service

```
apiVersion: v1
kind: Service
metadata:
  name: sip-loadb
  labels:
    app: sip-proxy
    project: qos-based-scaling
spec:
  ports:
    - name: sip-loadb
      protocol: UDP
      port: 5060
      targetPort: 5060
    - name: sipmetrics
      protocol: TCP
      port: 9494
      targetPort: 9494
    - name: analyzer
      protocol: TCP
      port: 9101
      targetPort: 9101
  selector:
    app: sip-loadb
  clusterIP: {{.Values.Containers.SipLoadB.clusterIP}}
  type: ClusterIP
  sessionAffinity: None
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
  internalTrafficPolicy: Cluster
```

A.11 SIP Configmap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sip-proxy-configmap
data:
  {{- $files := .Files }}
  {{- range $key, $value := .Files }}
  {{- if hasPrefix "configs/sip-proxy/etc/kamailio/" $key }} {{/* only when in configs/sip-proxy/etc/
     kamailio/ */}}
  {{ $key | trimPrefix "configs/sip-proxy/etc/kamailio/" }}: {{ $files.Get $key | quote }} {{/* adapt
      $key as desired */}}
  {{- end }}
  {{- end }}
```

Bibliography

[1] Cisco. Cisco annual Internet report (2018 - 2023) white paper. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, 2022. Accessed: 2022-09-14.

[2] Qiang Ye, Junling Li, Kaige Qu, Weihua Zhuang, Xuemin Sherman Shen, and Xu Li. End-to-end quality of service in 5G networks: Examining the effectiveness of a network slicing framework. *IEEE Vehicular Technology Magazine*, 13(2):65–74, 2018.

[3] 3GPP. IP Multimedia Subsystem (IMS); Stage 2. Technical Specification (TS) 23.228, 3rd Generation Partnership Project (3GPP), 12 2021. Version 17.3.0.

[4] 3GPP. Requirements for Evolved UTRA (E-UTRA) and Evolved UTRAN (E-UTRAN). Technical Report (TR) 25.913, 3rd Generation Partnership Project (3GPP), 12 2009. Version 9.0.0.

[5] 3GPP. Study on technical aspects on roaming end-to-end scenarios with Voice over LTE (VoLTE) IP Multimedia Subsystem (IMS) and other networks. Technical Report (TR) 29.949, 3rd Generation Partnership Project (3GPP), 3 2022. Version 17.0.0.

[6] A. U. Rehman, Rui L. Aguiar, and João Paulo Barraca. Network functions virtualization: The long road to commercial deployments. *IEEE Access*, 7:60439–60464, 2019.

[7] 3GPP. System architecture for the 5G System (5GS). Technical Specification (TS) 23.501, 3rd Generation Partnership Project (3GPP), 9 2022. Version 17.6.0.

[8] Cloud Native Computing Foundation. CNCF: Kubernetes. https://www.cncf.io/projects/kubernetes/. Accessed: 2022-10-22.

[9] Nokia. Nokia launches cloud-native IMS voice core product to simplify network operations for communication service providers. https://www.nokia.com/about-us/news/releases/2022/05/30/nokia-launches-cloud-native-ims-voice-core-product-to-simplify-network-operations- Accessed: 2022-09-14.

[10] Abderaouf Khichane, Ilhem Fajjari, Nadjib Aitsaadi, and Mourad Gueroui. Cloud native 5G: an efficient orchestration of cloud native 5G system. In *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2022.

[11] Cloud Native Computing Foundation. Kubernetes: Horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2022-10-22.

[12] Cloud Native Computing Foundation. Prometheus: Overview. https://prometheus.io/docs/introduction/overview/. Accessed: 2022-10-22.

[13] Csaba Rotter and Tien Van Do. A queueing model for threshold-based scaling of UPF instances in 5G core. *IEEE Access*, 9:81443–81453, 2021.

[14] Hai T. Nguyen, Tien Van Do, and Csaba Rotter. Scaling upf instances in 5G/6G core with deep reinforcement learning. *IEEE Access*, 9:165892–165906, 2021.

[15] Cloud Native Computing Foundation. Kubernetes: Assigning pods to nodes. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#nodeselector. Accessed: 2022-10-12.

[16] Kubernetes: Statefulsets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. Accessed: 2022-10-12.

[17] Cloud Native Computing Foundation. Kubernetes: Deployments. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Accessed: 2022-10-22.

[18] Kubernetes: Service spec. https://kubernetes.io/docs/reference/kubernetes-api/service-resources/service-v1/#ServiceSpec. Accessed: 2022-10-12.

[19] DockerHub. Dockerhub: Kamailio. https://hub.docker.com/r/kamailio/kamailio. Accessed: 2022-10-22.

[20] Nick. Kamailio. https://nickvsnetworking.com/category/voip/kamailio/. Accessed: 2022-11-01.

[21] Eldadru. Eldadru/ksniff: Kubectl plugin to ease sniffing on kubernetes pods using tcpdump and wireshark. https://github.com/eldadru/ksniff. Accessed: 2022-10-22.

[22] ghettovoice. gosip.

[23] Cseh Viktor. Kalbi golang session initiated protocol framework.

[24] hyperioxx. Kalbi golang session initiated protocol framework.

[25] Cloud Native Computing Foundation. Kubernetes: Assign cpu resources to containers and pods. https://kubernetes.io/docs/tasks/configure-pod-container/assign-cpu-resource/. Accessed: 2022-10-22.

[26] Redis. https://redis.io/. Accessed: 2022-11-01.

[27] Gianni Barlacchi, Marco De Nadai, Roberto Larcher, Antonio Casella, Cristiana Chitic, Giovanni Torrisi, Fabrizio Antonelli, Alessandro Vespignani, Alex Pentland, and Bruno Lepri. A multi-source dataset of urban life in the city of milan and the province of trentino. *Scientific Data*, 2(1):150055, Oct 2015.

[28] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Sip: Session initiation protocol. RFC 3261, RFC Editor, June 2002. http://www.rfc-editor.org/rfc/rfc3261.txt.